Classifier Systems
for
Situated Autonomous Learning

Gary Allen Roberts

Ph.D.

University of Edinburgh

1991

Abstract

The ability to learn from experience is a key aspect of intelligence. Incorporating this ability into a computer is a formidable problem. Genetic algorithms coupled to learning classifier systems are powerful tools for tackling this task. While genetic algorithms can be shown to be near optimal solutions for the search task they perform, no similar proof exists for classifier systems.

My research investigated two aspects of classifier systems, classifier selection and credit assignment. Explicit world models, look ahead and incremental planning are incorporated into the classifier system framework in order to make use of more of the information available to the system, and a more sophisticated approach to credit assignment is attempted.

The investigation involved the construction of four different classifier systems, and testing each of these systems in three separate virtual worlds. Wilson's Animat research was carefully reconstructed, and used as the control in a scientific experiment testing the efficacy of the various strategies embodied in three experimental systems.

The three experimental classifier systems all contained explicit world models and lookahead. One was an extension of Wilson's Animat, the other two involved an entirely new credit assignment scheme inspired by Watkins's Q-learning technique. Use of this technique enabled the incorporation of an incremental planner, similar to Sutton's Dyna-Q research, into one of the classifier systems, distinguishing it from the other Q-learning based classifier system.

The research shows that use of explicit world models and lookahead significantly decreases the time required in order to discover paths to well rewarded goals. It also shows that incremental planning can be used to further increase learning speed.

While the experimental classifier systems were quick at discovery, they did not necessarily exploit these discoveries. Because of this, the performance of these sys-

tems was variable between different virtual worlds. The experimental systems were outperformed by the control in two out of the three virtual worlds investigated.

Nevertheless, the value of explicit world models and lookahead within classifier systems is established. Likewise, the adaptation of Q-learning and incremental planning to classifier systems has been successfully demonstrated.

Acknowledgements

I would like to thank my supervisors, past and present, for their help and guidance: Henry Thompson, Chris Thornton and Roberto Desimone. I would also like to thank Stewart Wilson for his aid in identifying some of the differences between his work, and earlier versions of my reconstruction. Richard Sutton was very helpful in increasing my understanding of Q-learning and Dyna-Q.

Richard Caley has been indispensable. Not only has he helped in shaping many of the ideas presented here, but his technical expertize and support have been unfailing. The many graphs in chapter 7 were produced using software at his disposal.

I must thank my family for their continuing support, particularly my father, Martin Roberts, and my brother, Jefery Roberts.

Several of the other Ph.D. candidates were also especially helpful and supportive. In particular, Colin Williams, Carla Gomes and Mike Cameron-Jones made important contributions.

Comments on several draft chapters by Rachel Weiss were much appreciated and many corrections were made based upon her efforts.

There are many others who have been important in making this thesis possible. My debt to them is deeper for not being able to include them all by name. The flaws remaining in this work are my responsibility alone.

The equipment used for this research was donated by the Rank Xerox Ltd. University Grants Programme.

Declaration

I composed this thesis myself, describing my own research.

Gary Allen Roberts

Table of Contents

List of Tables and Figures

Chapter 1

Introduction

1.1 Introduction

Learning is a crucial aspect of intelligence. One of the most difficult forms of learning [Carbonell *et al.* 1983] is autonomous learning, where no supervision or training scheme is available to the learner, yet this form appears to be universal in organisms that have any capacity to learn at all. It is therefore an important task within the field of artificial intelligence (AI) to understand and duplicate this capacity for autonomous learning. Specifically, I am interested in the problem of autonomous learning for a computer controlled automaton situated within an environment, real or simulated. This is the problem of situated autonomous learning[1]. In many ways, genetic algorithms are ideal tools for this task. Genetic algorithms are search algorithms based on concepts from modern genetic theory. They are capable of originating novel and plausible solutions, using information from the relative success of previous attempts. They do not require extensive knowledge of the domain to be searched, and have been used successfully for finding minima or maxima of complicated mathematical functions. In the case of function optimization, the solution space can be seen as a search through the space of parameters over which one is attempting to optimize. Unfortunately, in situated

[1]This has been called "The Animat path to AI" by Wilson [1990].

autonomous learning, the solution space we wish to search is not obvious. Also, genetic algorithms are sensitive to the representation of the solution space, so the amenability of the representation is an important issue. One candidate that has enjoyed reasonable success has been drawn from the concept of rule based systems.

Rule based systems have been used, with reasonable success, to choose a course of action based upon the situation (*e.g.* prescribing a particular antibiotic from the symptoms and history of a patient). Choosing the appropriate action for a given situation is exactly the problem facing an automaton situated within an environment. Further, rule based systems are equivalent in computational power to the Turing machine. Thus, a rule based system can perform any computation that can be executed on a digital computer. In order to harness these capabilities within the framework of genetic algorithms, a simple form of rule based system, called a classifier system, has been developed [Holland and Reitman 1978, Goldberg 1985, Smith 1983, Holland 1983]. The rules in a classifier system (such rules are referred to as classifiers) must conform to a strict simple syntax. This allows classifiers in the classifier system to be generated, reproduced, modified, and possibly replaced by operation of the genetic algorithm. The intent is to produce a system with the search power of genetic algorithms and the computational power of a rule based system.

In order for a classifier system to function properly, the relative utility of each classifier must be evaluated. This evaluation is a difficult problem, known as the credit assignment problem. My work involves a major departure from the standard methods for classifier evaluation. Most systems base the evaluation on attempting to assign responsibility for some distant future situation, that is more or less beneficial. My evaluation criteria is based on an explicit world model. The classifier system makes use of this model to predict the effect, and therefore the utility of each classifier. The information allowing the construction of such a model is ignored by current systems. A system making use of this additional information should be better able to outperform (and certainly do at least as well as) systems that do not make use of this information. Realizing this advantage, however, is not a trivial matter. First, a model must be built of the effect of the classifiers

within the system. Secondly, this model must be reasonably used to modify the operation of the classifier system and the genetic algorithm.

Creating an explicit world model is fraught with problems. The world is an ambiguous place, where doing what appears to be the same action in what appears to be the same place does not of necessity produce the same result. There may be hidden variables that the system does not detect, or, as is the case of the models of our universe produced by modern physicists, the world we wish to model may not be deterministic. The universe defies the ability of human intelligence to build a correct comprehensive model. We should not expect our artificial intelligence to do better.

Utilizing a world model can also be difficult. Depending upon the nature of the model, effectively searching through it may take an exponential amount of time. Also, while searching through the model, inconsistencies may be discovered[2], the model must be modified even as it is utilized.

While there is considerable theoretical justification for the use of genetic algorithms, the theoretical underpinnings of classifier systems are rather weak. This research therefore concentrates on attempts to improve the classifier system and problems more specifically related to genetic algorithms are ignored (*e.g.* premature convergence and scaling). The extraction and utilization of the most information available to the situated autonomous learner are central.

Current classifier systems are totally reactive. They have no explicit ability to look ahead or plan. The construction of a world model, and use of lookahead and incremental planning allow explicit use of information implicit in the functioning of previous classifier systems. Since classifier systems are a form of rule based system, an important issue is the control strategy for choosing which classifier or classifiers to activate. It is in this area that I have concentrated my efforts. The

---

[2]I consider this to be a fortuitous possibility, since this might allow correction of the model without requiring recourse to experimentation. It can also guide future experimentation in order to correct the model (but whether this happens in the systems I have devised is debatable).

availability of an explicit world model should allow a more effective activation strategy and improve system performance.

One of the problems not addressed in this thesis is that of improvements to the genetic algorithm itself. In fact, the interface between the classifier system and the genetic algorithm seems to be a weak area.

1.2 Research Methodology

In the short history of genetic algorithms, there have been a plethora of techniques used. Even within the subfield of classifier systems, many differing implementations exist. It has become difficult to compare one's results with others, and to determine the cause of actual performance differences. In order to solve this problem, it seemed that the framework of a rational reconstruction [Bundy 1986] would be the best approach. Rational reconstruction involves constructing a common framework within which to study the differing systems, both past and present. One must duplicate the previous works which one wishes to compare with the current system. To that end I attempted a rational reconstruction encompassing two systems described in the literature: Wilson's Animat and the CS-1 system of Holland and Reitman. A large amount of time and effort was expended attempting to get the reconstructed systems to perform as documented [Roberts 1989], however, the reconstructions did not match the performance of the originals. In the case of the Holland and Reitman system, the differences were so pronounced that CS-1 was dropped from the comparison group.

Three virtual worlds were constructed in order to compare the performance of the classifier systems being investigated. One was taken from Wilson's original work, one was a modification of Wilson's virtual world, and the third was a virtual world designed specifically to show off the advantages of lookahead. As part of the rational reconstruction, a statistical evaluation scheme was devised to allow a fairly objective comparison of results. This allows a fairly unbiased relative evaluation of each system.

1.3 Design Philosophy

When designing and implementing a computer program, various choices must be made. There is often a tendency to become involved in programming details, and to forget the higher goals when making these decisions. When this happens, the results tend to be internally inconsistent. Worse yet, the goals originally desired may not be obtained, making the entire effort useless. It is therefore important from time to time to take a step back, and examine the goals. In fact, it is useful to take a step even further back to determine why the goals are deemed important. Current decisions can then be made in the light of the priorities established.

A great deal of effort in the field of artificial intelligence is spent attempting to simulate higher order human expertise. While this has met with some success, the resulting systems tend to be brittle in the sense that a system cannot effectively use its knowledge to solve problems similar but not identical to those for which it was programmed.

Brooks [1986] criticizes current research in Artificial Intelligence on the basis that it does not attack the appropriate problem. Current systems tend to deal with abstractions and higher level cognitive reasoning. Brooks argues that the abstractions used are likely to be erroneous constructions based only upon introspection, rather than anything like what we actually use internally. Even if, by some chance, the abstraction is useful, the interesting part of the intelligence problem is making the abstraction, not just using it. He also argues that higher level cognitive reasoning, as a very recent phenomenon in the evolutionary history of life on Earth, is a relatively trivial addition to the basic problems of moving, sensing, and reacting in a dynamic environment.

Holland [Holland and Reitman 1978] has demonstrated a system (CS-1) that simulates lower order, animal-like learning, within simple simulated environments. Such a system with experience in one environment learned much more quickly a similar environment, than one with no experience.

Brooks criticizes what he refers to as "toy worlds" and appears to consider any software simulated environment to fall under that classification. Toy worlds,

according to Brooks, are those "where it is not up to the AI system itself to do all the understanding of the world itself without relying on a human interpreter". Similarly, it is a toy world "if the AI system is not responsible for carrying out its actions in the world without a human agent to interpret its responses". Brooks observes that special environments designed for robotics research can also be toy worlds, certainly in those cases where the environment is assumed to be static.

While I find Brooks's criticism telling, I find it useful to distinguish where an approach fails, rather than insisting that all attempts satisfy all the various criteria. Use of simulated environments allows repeatability and control over the testing of candidate algorithms. It is not so important to learn that my algorithms perform poorly if they are controlling a real world robot under real world conditions (as I expect they would). It is much more important to determine why they perform as they do, and what factors influence their performance. Objective testing in the definable terrain of a simulator is invaluable. The virtual worlds used in my research are flawed in numerous ways. They are static and discrete. Still, they can provide evidence for how the controlling programs would perform in a non-static or non-discrete environment[3].

From the considerations of the previous section, and from the ultimate goal of achieving a real world artificial intelligence, I emphasize two aspects of the programming problem. Since any world model, implicit or explicit, within a real robot will be contained in the world, this world model will be less complex than the world and, therefore, incorrect. Since things happen dynamically, regardless of the state of the robot, the robot must always be ready to react. Speed and efficiency become important considerations.

If some internal world model could be correct, then there would be no point in learning after the correct model had been achieved. In fact, it is quite common in AI research to assume that the correct model can be programmed in, and learning is superfluous. When one considers that the internal world model cannot be known

---

[3]If a digital computer is to be included as part of the autonomous learning system, the signals being processed will be discrete in any case.

to be correct even if the world being explored is finite state [Gold 1967], it becomes clear that learning is a fundamental part of intelligent behaviour throughout its existence. This is why learning is an intrinsic part of the system I have designed.

Given real-time constraints, the system must be computationally efficient, which also puts limits on the amount of memory required. In a real world artificial intelligence, it would be the actual run-time speed that is at issue in determining whether its reaction time is sufficient. In my simulations, actual run-time information was difficult to obtain. The theoretical complexity of classifier systems, however, is quite promising. A classifier system can be made to run in time linear to the number of classifiers in the system. The time required is also linear on the number of features detectable by the system. Given the current state of computer hardware design, it is instructive to look at the multiprocessor theoretical timing as well as the single processor figures. Classifier systems can make extensive utilization of parallel processing[4].

1.4 Thesis Overview

Chapter 1 – Introductory Material

This chapter introduces the concept of autonomous learning, and how genetic algorithms may be used as a mechanism for autonomous learning. Some difficulties associated with genetic algorithms are discussed in a general way, and why these have led to the introduction of the classifier system.

Chapter 2 – Description of Genetic Algorithms

This is a general introduction to genetic algorithms, including some historical information as well as some of the theoretical justification for the success of genetic

---

[4]Classifier systems can be made to work like connectionist networks, so advantages there can be claimed in classifier systems as well [Davis 1989, Belew and Gherrity 1989].

algorithms. An exemplary genetic algorithm is discussed and the results of my experimentation with this system are presented.

Chapter 3 – Description of Classifier Systems

This chapter includes a description of classifier systems, and why it was necessary to introduce them. Two example classifier systems are discussed, and the results of my experimentation with these systems are presented.

Chapter 4 – Rational Reconstructions

This includes the results of reconstructions I made of Wilson's Animat, and Holland and Reitman's CS-1. The results obtained differ markedly from the published material.

Chapter 5 – The Problem Domain

This chapter describes the kind of problems we would like to be able to solve, and characterizes the important features. This chapter also provides much of the motivation for changing the apportionment of credit system currently used in many classifier systems.

Chapter 6 – Look Ahead Classifier Systems

This chapter describes attempts to solve the problems presented in chapter 5. This chapter includes three classifier systems that I have improved to have the capacity to look ahead. One is a modification of Wilson's original research, and has many similarities. The other two systems employ a new control mechanism that allows information from multiple classifiers to be combined to provide more accurate assessment of situations encountered or predicted. They also incorporate an alternative mechanism to the method of credit assignment inspired by Watkins's Q-learning research. One of these systems takes advantage of this new credit assignment system to allow incremental dynamic planning in similar fashion to Sutton's Dyna-Q learning system. Such a system can be expanded to incorporate any planning scheme desired.

Chapter 7 – A Statistical Comparison of the Results

This chapter presents a comparison between the systems I have devised and one of the systems I have re-implemented from the literature, on the problem domains specified in chapter 5.

Chapter 8 – Limitations of the System (Future Work)

This chapter investigates some of the limitations of the systems described, and suggests possible ways of getting around these limitations

Chapter 9 – What Has Been Accomplished?

This chapter sums up the accomplishments of my research to date.

Chapter 2

Description of Genetic Algorithms

2.1 Introduction

While the focus of my work has been in the area of classifier systems, and to a large extent independent from genetic algorithms research, it really is not possible to discuss classifier systems without an understanding of the genetic algorithms that drive them. Here then is a brief look at the history of these systems, and how they work.

Darwin expounded the principles of evolution via variant offspring and natural selection in 1859. The application of these ideas to programming dates back to the 1950's. By the 1970's, however, initial interest in this technique within the field of artificial intelligence had turned to rejection [Goldberg 1989a]. Few researchers continued along those lines. In 1975, however, the publication of *Adaptation in Natural and Artificial Systems* [Holland 1975] gave new impetus to genetic algorithms research. Holland provides new formal genetic operators, based upon modern genetic theory, and a mathematical framework explaining why these operators should be effective.

The most important distinction between Holland's system of genetic algorithms, and its predecessors that relied primarily on mutation and selection, is an analogue to the recombinant genetics of biological evolutionary systems, the crossover operator. Holland shows that this operator invokes a powerful implicitly parallel search strategy (see section 2.3).

In order to use a genetic algorithm (or any other form of evolution based programming) one needs a problem domain, and must create a search space for potential solutions that is amenable to the genetic operators to be employed in searching this space. Once these are chosen, the algorithm works as follows:

1) Initial input consists of items from the search space (perhaps generated at random). This is the initial population.

2) Each member of the population is evaluated on some criteria, normally based upon how good a solution this member is in the problem domain (in biological terms, the fitness of the member). The algorithm may be halted based on the evaluation, if a good enough solution has been found, otherwise, step 3 is performed.

3) A new population is created consisting of allowed genetic operators on members of the previous population, and step 2 is performed[1].

The power of modern genetic algorithms lies in the representation of the population members and the type of genetic operations possible. Both of these are adapted from modern genetic theory. Together they provide a provably efficient search strategy.

Elements in the search space are represented as fixed length vectors. Each position in the vector (normally representing a parameter in the search space) has a set of legal values (It has been shown theoretically [Holland 1975] that the sets should be small, although Antonisse [1989] challenges this claim.).

In the genetic algorithm, we can see data structures being interpreted in two different ways. In step 2, each element of the population is evaluated by some criteria. This is equivalent to the phenotype of biological systems. In step 3, the interpretation made in order to evaluate the population is unimportant (except in

---

[1]In a standard genetic algorithm this means that the entire population changes with each iteration of the genetic algorithm. Such a significant change in the population can be undesirable. Often (*e.g.* in most classifier systems), only a small portion of the population is chosen to be replaced.

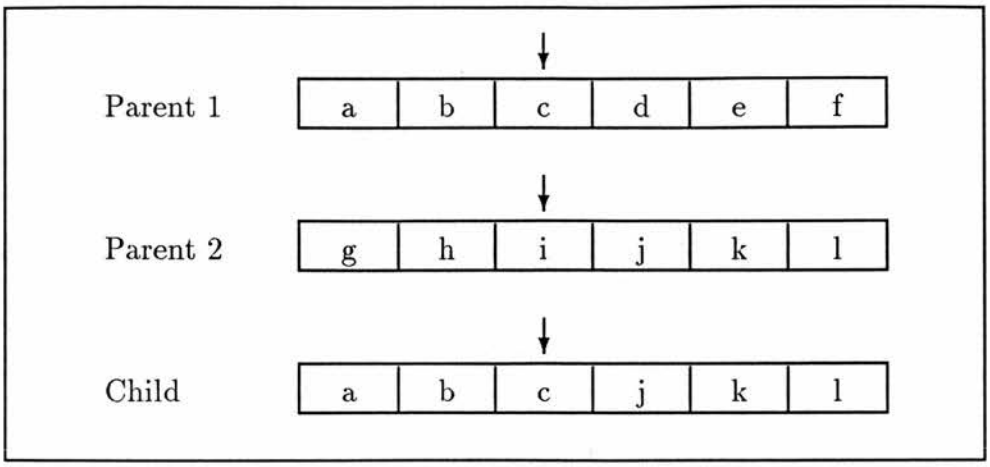

Figure 2–1: Single Point Crossover

so far as what fitness it assigns to these structures). The member of the population is treated as a vector subject to the genetic operators of the system. This is the equivalent to the genotype or genome of biological systems.

In modern genetic algorithms, the most important genetic operator is a recombination of two parent forms. This operator is known as crossover. There are three crossover operators that are most common in the literature: single point, two point, and uniform crossover. Holland analyzed single point crossover. Two point crossover avoids end effects. Uniform crossover was designed to avoid biases based upon the length of the fixed length vectors, and also avoids adjacency effects. Each has its own proponents. There are many other schemes as well.

In single point crossover (figure 2–1), a random position is picked in the vector representation (not the last position). The child consists of one parent copied up to and including the chosen position, and the rest copied from the other parent. While this may not seem *a priori* to be a particularly useful operator, Holland [1975] has shown that it can be used to implement a massively parallel search of the hyperplanes defined by the population vectors (this is explained in section 2.3).

In two point crossover (the form most used in my research), the vectors of the search space can be thought of as rings, with the first element of the vector considered to follow the last element in the vector. Two cut points are chosen

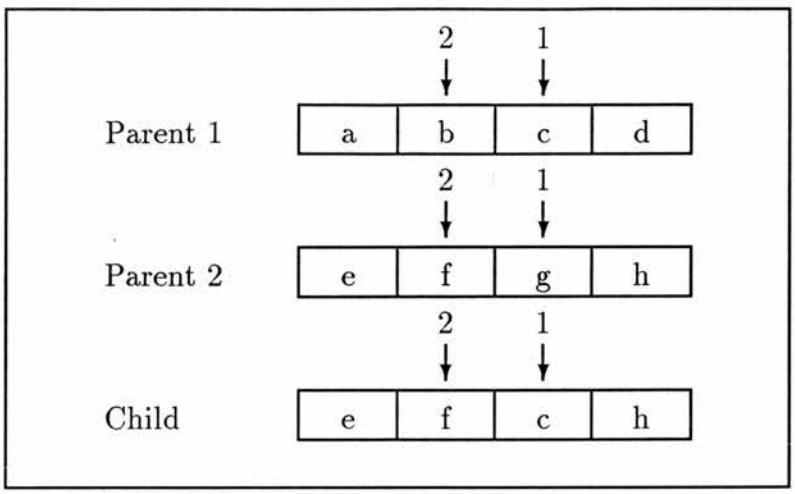

Figure 2–2: Two Point Crossover (linear representation)

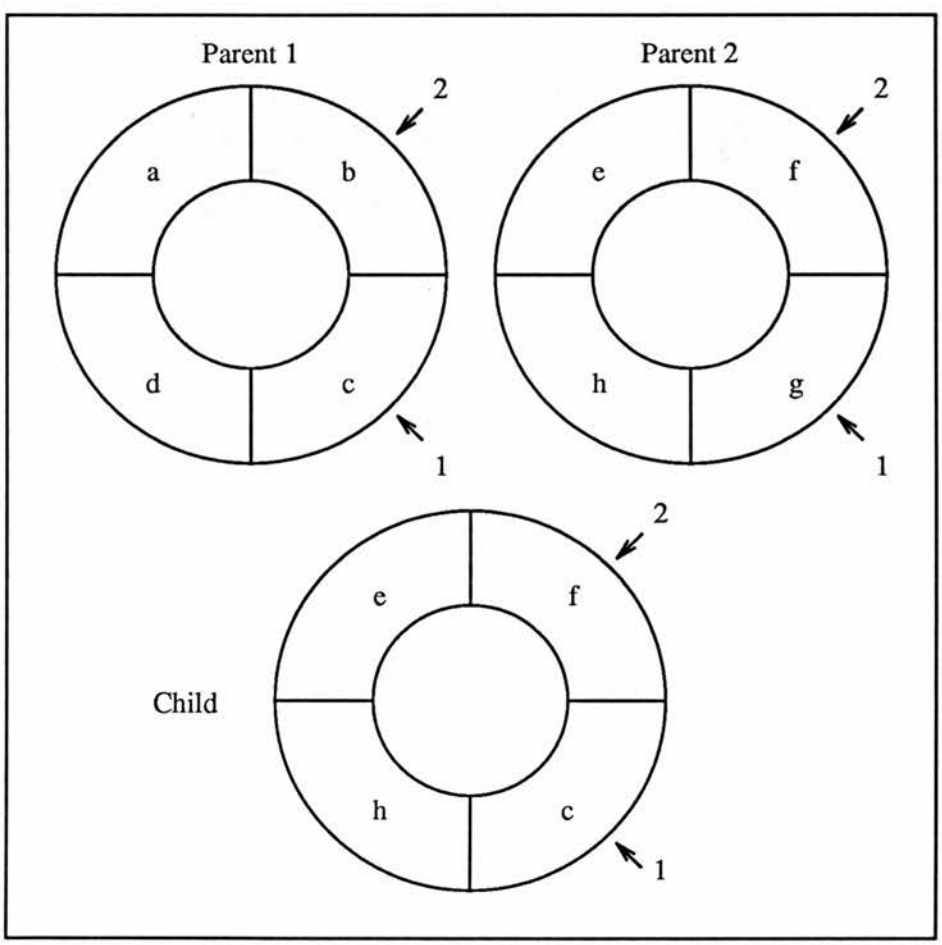

Figure 2–3: Two Point Crossover (circular representation)

(they should not be the same point), the offspring consists of a copy of the second parent, moving clockwise from just after the first cut point, up to, and including the second cut point, and the rest copied the first parent, from just after the second cut point, up to, and including the first cut point (see figure 2–2 for the linear representation, and figure 2–3 for the circular interpretation). Single point crossover can be seen as two point crossover where the first crossover point is always equal to the length of the vector.

By extension, one can see that it is possible to have n point crossover, for any n between one and one less than the length of the population vectors. Rather than choose a particular n, there is also the possibility of uniform crossover.

In uniform crossover the probability of crossing over at any particular point is some constant. The child consists of the first item from one of the parents, and at each point that the crossover probability is met, the parent from that point on is swapped, until again the probability of changing back is met.

Some researchers produce two offspring from each crossover operation, the second offspring having the same cut points as the first, but the parents are swapped. In my research crossover is considered to produce exactly one offspring.

With only mutation in a genetic algorithm, it is simply a case of generate and test. Crossover provides a technique for a generate plausible and test, in a way that is applicable to a large number of problem domains. A brief discussion of why crossover produces plausible results is given in section 2.3.

2.2 A Genetic Algorithm - Dewdney's AUTOSOUP

When using a genetic algorithm, one data structure is interpreted in two ways. One way is as something to be manipulated by genetic operators, and the other is as something to be evaluated. An example from the popular literature [Dewdney 1985] may prove illustrative. In order to use a genetic algorithm, one must have a representation amenable to the genetic operators, and an interpretation of this genotype to produce an evaluation function. Dewdney[2] conceives of a stream of bits for which he would like to evolve a predictor. That is, given the beginning of a binary input string $b_0, b_1, \ldots, b_n$, predict the value of the next item in the string, b_{n+1}. Perfect prediction is impossible unless restrictions are made on the possible input strings.

Dewdney chooses finite state transducers, with binary alphabets as both input and output alphabets, as the predictors of the search space. Because of this choice, the only strings that can be perfectly predicted are those that repeat after some point. For such transducers, the longest a string may be before repeating, and be correctly predicted, is just twice the number of states in the transducer. In fact, in an n state transducer, there are strings of length $n + 2$ that cannot be accurately predicted.

In order to maintain a fixed-length vector representation, Dewdney fixes the number of states in his transducers, initially discussing three-state transducers, while his implementation searches the space of four-state transducers. Thus it can be seen that the choice of problem domain, search space, and representation are linked. If perfect predictors are required, the designated search space limits the problem domain to repeating strings of a particular form.

---

[2]Actually, the problem domain, solution domain and the encodings were taken from work by Fogel *et al.* [1966]. Aside from the popularization, Dewdney's contribution was to try crossover in this domain. Fogel had a different genetic operator for combining two parental forms to produce offspring, but Dewdney excludes even the mention of this from his article.

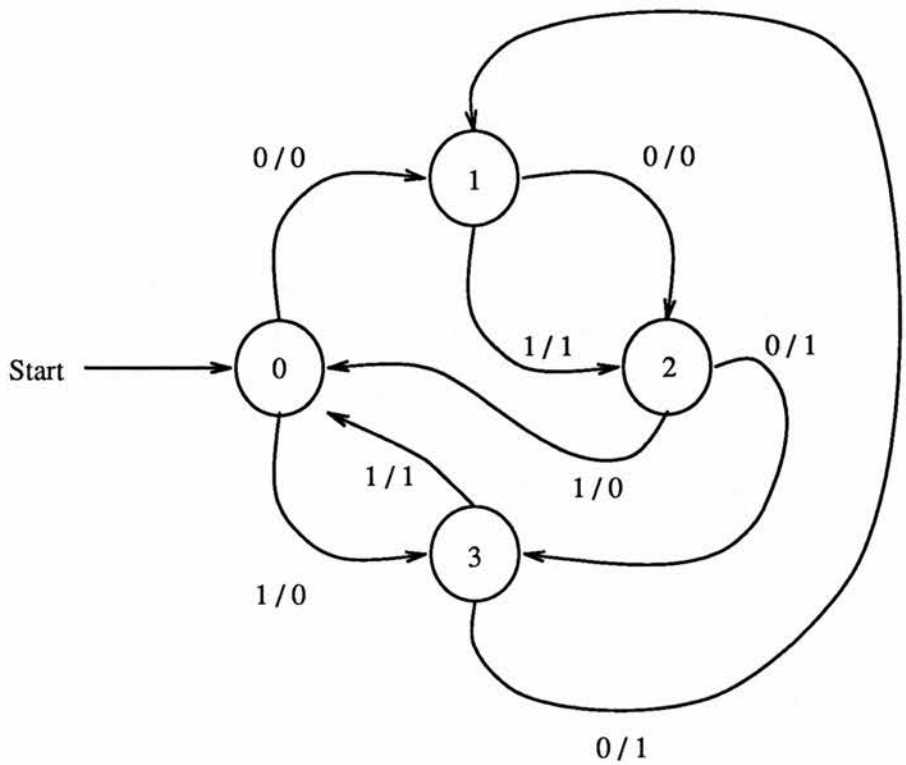

Figure 2–4: A Four-State Transducer—01:03||02:12||13:00||11:10

The encoding used by Dewdney is a vector containing four elements for each state in the transducer. The first two elements represent what the transducer should output, and what state the transducer should enter (respectively), on receiving a zero input in that state. Likewise, the next two elements determine the output and state change to perform if the automaton receives a one as input.

An example four-state transducer in vector notation might be written as: "0103021213001110". Such notation is difficult for people to decipher (see figure 2–4 for a transition diagram[3]), but if we use "||" to separate states, and ":" to separate the consequence in that state for a 0 input versus a one input, it is fairly simple to interpret. "01:03||02:12||13:00||11:10" is a genotype for a transducer that in state 0 (the initial state) will, upon receiving a 0 input, predict a 0 to follow

[3]The labels on the arcs are *input / prediction* pairs (*e.g.* 1 / 0 means: on a 1 as input, take this arc and predict a 0 will follow).

and move to state 1. Having moved to state 1, should it receive the predicted 0 from the environment, predict another 0 and move to state 2). This transducer is a perfect predictor for the repeating string $\overline{00011011}$. It is by no means the only perfect predictor for $\overline{00011011}$. There are 144 perfect predictors for that repeating sequence out of the 16777216 $(= (2 \times 4)^8)$ possible genotypes representable in the vector notation specified. Also, since the labels of the states do not affect the functioning of the transducer, there are at least six genotypes for each phenotype (the initial state label is important, and cannot in general be changed, but there are three possible names for state 1, and two possible names for state 2, leaving the name for state 3 fixed).

Dewdney starts with a randomly generated population of 10 vectors (step 1 in the explanation of a generic genetic algorithm). Each of the vectors is rated in the following manner. An environmental sequence of 100 bits is presented to each transducer represented in the population (note that it is the phenotype that is tested, not the genotype). The transducer scores 1 for every correct prediction, and 0 for every incorrect prediction. If a transducer has a score above the threshold for termination, the program terminates (step 2 from the explanation of a generic genetic algorithm). If no transducer in the population is determined to have a sufficiently high score, a genetic operation is performed (step 3 in the explanation of a generic genetic algorithm).

There are two genetic operators used in AUTOSOUP, a modified two point crossover, and an incremental point mutation. In each iteration of the genetic algorithm there is a certain chance of crossover (0.25 in my reconstruction) between the top scoring genotype and a random member of the population (perhaps the same top scoring genotype). The difference between Dewdney's crossover, and standard two point crossover, is that Dewdney never selects a crossover with the second cut point before the first. The result of such a crossover replaces the lowest scoring member of the population. Whether crossover has occurred or not, a mutation operation is applied to a randomly selected member of the population. This mutation operator is applied directly, and destructively, to the vector

selected. Rather than randomly changing the selected allele[4] of a population vector, Dewdney uses the numerical ordering inherent in his representation. The site chosen is incremented in value, and if it exceeds the values permissible at that site, the site is given the value 0. All vectors unaffected by these genetic operators continue into the next generation (note that this gradual change in the population is contrary to the standard "generational" approach used in genetic algorithms, but is quite common in the classifier systems that will be discussed later).

My reconstruction of Dewdney's AUTOSOUP program was able to find a perfect predictor of the string presented above after an average of 9690 trials over 100 runs with a standard deviation of 7857. By way of comparison, note that random search would give an average of 116508 (= 16777216/144) trials to find a perfect predictor for the string in question.

It should be noted that Dewdney has made some unusual choices in his AUTOSOUP program, but genetic algorithms are quite robust, so performance is not necessarily impaired. Dewdney claims that his incremental mutation operator is "random enough". To test this, I substituted random mutation for Dewdney's incremental mutation in my reconstruction. The average number of trials required over 100 runs dropped to 9016 with a standard deviation of 7201. This change is not statistically significant. Experience with genetic algorithms suggests that increased use of crossover, and less reliance on mutation (perhaps with a larger population size), would improve performance, but I have not performed these experiments.

2.3 Theoretical Aspects of Genetic Algorithms

Crossover has been shown empirically to be a very powerful genetic operator. There is theoretical justification for this in what is known as the schema theorem[5],

---

[4]An allele is a gene occupying a specific site.

[5]There have been empirical results that indicate there are other reasons as well for the success of genetic algorithms.

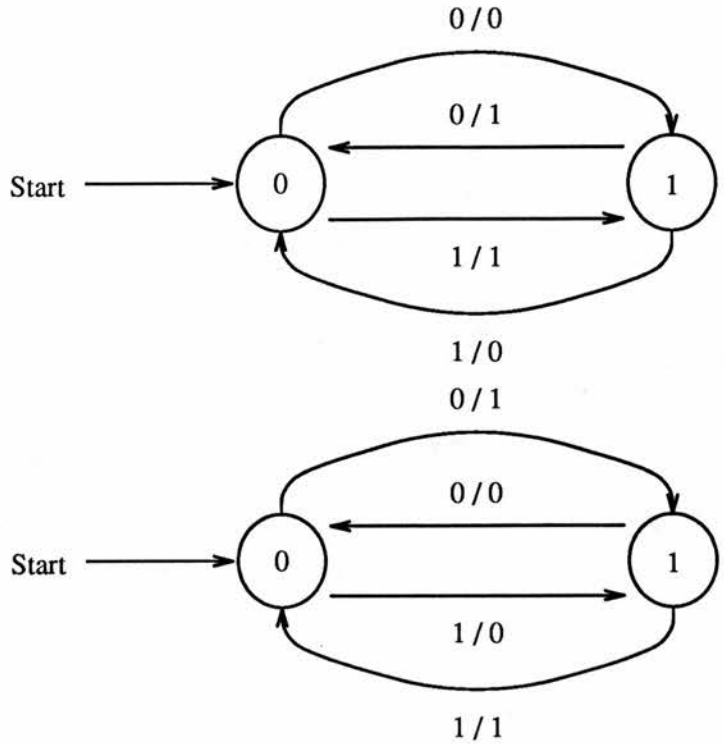

Figure 2–5: Two Two-State Transducers—top–01:11||10:00, bottom–11:01||00:10

which I shall describe shortly. First, let us consider two genomes for different two-state transducers, based upon the notation of the previous section:

$$01{:}11\|10{:}00$$
$$11{:}01\|00{:}10$$

These transducers are shown in figure 2–5. While these two transducers predict different strings, it can be seen that there is structural similarity. This similarity is reflected in the genotypes. Specifically, if one looks at corresponding sites on each genome, there are certain common genes. If we use an asterisk "*", as a placeholder for those sites not of current interest, we can show the commonality of the two genomes as "$*1{:}*1\| * 0{:}*0$". Such a representation is referred to as a schema[6]. There are sixteen distinct genotypes for two-state automata with

[6]The "*" used in schemata is chosen to be extra-lingual to the alphabet of the genotype.

this schema. There are also sixteen schemata that are common between the two example genomes[7]. Each of the genomes here conform to 256 schemata (more generally, any genome of length n conforms to 2^n schemata)[8]. Since sixteen of these schemata are in common, the example two genotypes are representative of $2 \times 256 - 16 = 496$ schemata. The utility values associated with the genotypes can be used as an estimator for the average value of all schemata represented by those genotypes. As a genetic algorithm searches through populations of genomes, so too can this be viewed as a search through sets of schemata. As schemata space tends to be much larger than the population used in the genetic search, many more schemata tend to be examined than genomes. Holland terms this "implicit parallelism" [Holland 1975]. Whether this "implicit parallelism" is of use depends largely upon how well the genetic algorithm searches this schemata space. This problem has been investigated elsewhere [Holland 1975, Goldberg 1989a], and only a short summary will be reported here.

If one chooses a schema and then randomly creates genomes exemplary of that schema, each genome can be evaluated, and the fitness of the genome calculated. This will produce a fitness distribution characteristic of that schema, with a mean and standard deviation. Likewise, a different schema would produce a (possibly different) mean and standard deviation. If one wishes to maximize fitness, a choice must be made as to how often to try genomes matching one schema, as opposed to genomes matching the other. In the case of genetic algorithms, the mean and

---

[7]The coincidental equality between the number of genotypes represented by the schema, and the number of schemata in common between the two example genotypes has two causes. First, the number of placeholders and the number of specified values in the example schema are equal. Secondly, the genetic alphabet available for substitution for each placeholder happens to be binary. If our transducers were allowed to predict a "2" in the input, as well as "0" and "1", there would be 81 genomes conforming to the example schema. There would remain only 16 schemata common between the two example genomes, no matter what the cardinality of the genetic alphabet.

[8]Note that one of these schemata will have no "*" marking at all, and, except for the difference in type (schema versus genome), it will be identical to the genome.

standard deviation of the fitness distribution for the schemata are not known *a priori*. Otherwise one could just choose the schema with the highest mean fitness. Holland has calculated the optimum sampling probabilities. Unfortunately, the precise optimal sampling probability function is itself a function of the (unknown) means and standard deviations of the distributions being investigated. Still, the form of the function is clear. Slightly more than exponentially increasing trials should be allocated to the schema with the observed greatest mean fitness. This is true even when there are more than two schemata directly competing (*i.e.* those schemata that for each site on the chromosome, either each schema has a placeholder there, or each schema has a specific gene there).

Genetic algorithms with simple reproduction based upon fitness, that is, without mutation or crossover, give an exponentially increasing number of trials to the directly competing schemata (remember that there are many such competitions going on in parallel). This should give a near optimal allocation of trials.

Unfortunately, a genetic algorithm with only fitness based reproduction does not produce new individuals or investigate schemata not contained in the original population[9]. This is where mutation and crossover are important. However, both of these operators interfere with the near optimal sampling of schemata. Specifically, crossover tends to interfere with the optimal processing of schema where the distance on the chromosome between the defined alleles furthest from each other in the schema is great, and mutation interferes with the optimal processing of schemata with many defined alleles. This, in qualitative terms, is the schema theorem. Holland estimates that for a genetic algorithm with a population size of n, on the order of n^3 schemata are near optimally processed each generation.

Since the schema theorem gives a reason for the success of genetic algorithms, it also gives criteria for judging how well it might work on specific problems, given a particular encoding for the genomes. Since we have characterized the schemata that are properly processed via genetic algorithm, we also characterize

---

[9]Indeed, the estimation of the mean fitness for even the schemata present does not appear to be very well investigated.

those schemata that are not properly processed, those with a large distance between their defining alleles, and those with many defined alleles. For example a schema with no placeholders at all would not be properly processed, yet such a schema is exactly what we are searching for. The optimal schema is (ignoring the type change) also the optimal genotype. In order to explain why optimal genotypes are generated and remain in the population, we need something beyond the schema theorem.

What has been proposed is called "The Building Block Hypothesis". The idea behind this hypothesis is fairly simple. Schemata that are not guaranteed to be properly processed by the schema theorem are put together (via crossover) from the schemata that are properly processed. Given a problem domain and a representation, there are techniques for analyzing how easily building blocks for the larger schemata are maintained in the population. Bethke's doctoral dissertation in 1981 defined techniques, using Walsh coefficients [Goldberg 1989a, Goldberg 1989b, Goldberg 1989c] for analyzing the difficulty of maintaining the appropriate building blocks within the population, and work in this area continues[Bridges and Goldberg 1989]. There does appear to be a correlation between problems that are deceptive in the sense of this sort of analysis, and those that are indeed hard for the genetic algorithm to solve. Unfortunately, this analysis technique requires an examination of the entire search space in order to make accurate predictions. If one really were only interested in the optimal solution for the problem domain, there would be no reason to run the genetic algorithm after the analysis was made. The optimal solution would already have shown itself in the analysis phase. Also, there are many search spaces that are just too large to analyze (which is why one might have chosen a genetic algorithm to search the space in the first place). For example, I have never seen such an analysis performed on a classifier system.

In general, there has been much analysis of genetic algorithms, and one can use them with a fair amount of confidence. There have been numerous studies of how they work, and how best to solve many of the problems associated with their use. Very little of this analysis or indeed the empirical results has been, or can be easily applied to classifier systems. Nevertheless, it is classifier systems, which are introduced in the next chapter, that are the main topic of this thesis.

Chapter 3

Description of Classifier Systems

3.1 The Need for Classifier Systems

Genetic algorithms are philosophically based on the concept of chromosomes and genetic recombination. Just as the genes of chromosomes need to be decoded into enzymes in order to control the functioning of a cell, so the genotype that a genetic algorithm manipulates must be decoded in order to control the functioning of an automaton. Various encodings are possible (the generality of the encoding scheme is one of the important virtues of genetic algorithms). We have seen the interpretation of the genotype as a four-state transducer (see chapter 2). If we wish to tackle the most complex problems computable, the interpretation must be Turing equivalent.

Even in his early work on genetic algorithms [Holland 1975], Holland realized the need for some system with the power of a Turing machine, but amenable to the adaptation mechanism of genetic algorithms. He designed what he called a "broadcast language". This broadcast language has never been tested with genetic algorithms, but the scheme has many similarities to classifier systems.

The work on the first classifier system was published in 1978, by Holland and Reitman [Holland and Reitman 1978]. Since that time, a large number of systems have been termed classifier systems. These systems have varied widely, so that it is difficult to epitomize the archetypical classifier system. In this chapter, I attempt

a general description of classifier systems, and provide some details of archetypical systems.

3.2 The Core of a Classifier System

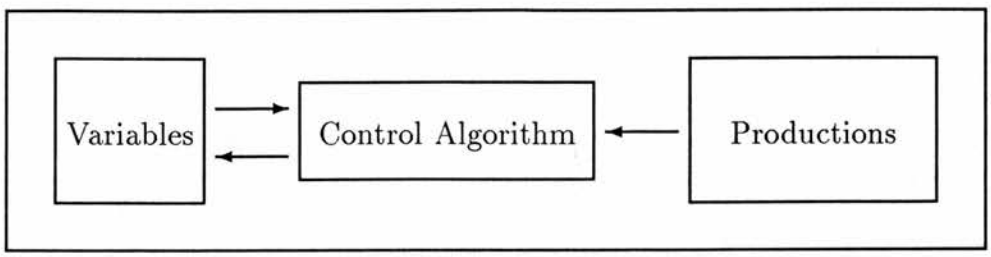

Figure 3–1: A Production System

All classifier systems have at their heart some form of production system that is amenable to modification via genetic algorithm. Goldberg [1989] calls the form of production system commonly used within classifier systems a "rule and message system" (henceforth RMS). If one considers a standard production system to consist of rules with preconditions and some database to manipulate in case the preconditions are satisfied, as well as a control algorithm that handles selection and execution of these rules (see figure 3–1), then its relationship to an RMS is straightforward.

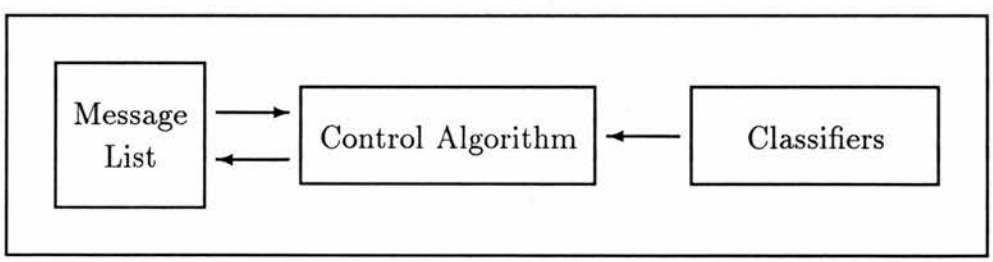

Figure 3–2: A Rule and Message System (RMS)

In a classifier system (and therefore in an RMS), the rules are known as classifiers (see figure 3–2). Rather than a database to manipulate, an RMS has a message list. The precondition of a classifier, known as a taxon, is satisfied when an appropriate message exists on the message list. Executing (otherwise known as activating) a classifier means placing some particular message or messages on the message list.

A message can be thought of as a context, a set of bindings for variables of potential pertinence to the system. The taxon of a classifier specifies the set of contexts over which the classifier may be activated. Thus, a message might contain the bindings { Pressure = 101325, Temperature = 5, Time = 10:21, DayType = WorkingDay }, and a taxon specifying contexts that match { Temperature < 16, 9:00 < Time < 21:00, DayType = WorkingDay } would be satisfied by such a message[1].

When the RMS is in operation, classifiers are chosen for activation based upon the current message list. Those classifiers with satisfied taxons are activated[2]. Each produces one or more messages which are all pooled to produce a new message list. The old message list is discarded. The process can then be repeated.

The above paragraphs describe the only two ways a classifier can interact with the message list. The taxon of the classifier can match one or more of the messages on the message list, in which case it is considered satisfied, and, upon activation by the system, it contributes one or more messages to the new message list.

Aside from the classifiers themselves, there are other processes that can contribute to and examine the message list (see figure 3–3). These are input and output processes (respectively). The sensors (or virtual sensors) of the automaton feed into an input processor, which converts this information into messages to be placed upon the message list. The output processor must control the effectors of the automaton, determining from the message list what action(s) to take. There is, however, a problem with this method of communication between the classifier system and its output processes.

If output processes had as simple a method of assessing the contents of the

---

[1]Note that in actual classifier systems, the encoding of information in the messages is determined by the dynamic functioning of the system and may be difficult to decipher intelligibly.

[2]This will change later on when there will be a need to have classifiers compete in order to be activated. Those classifiers with satisfied taxons will be eligible to enter this competition.

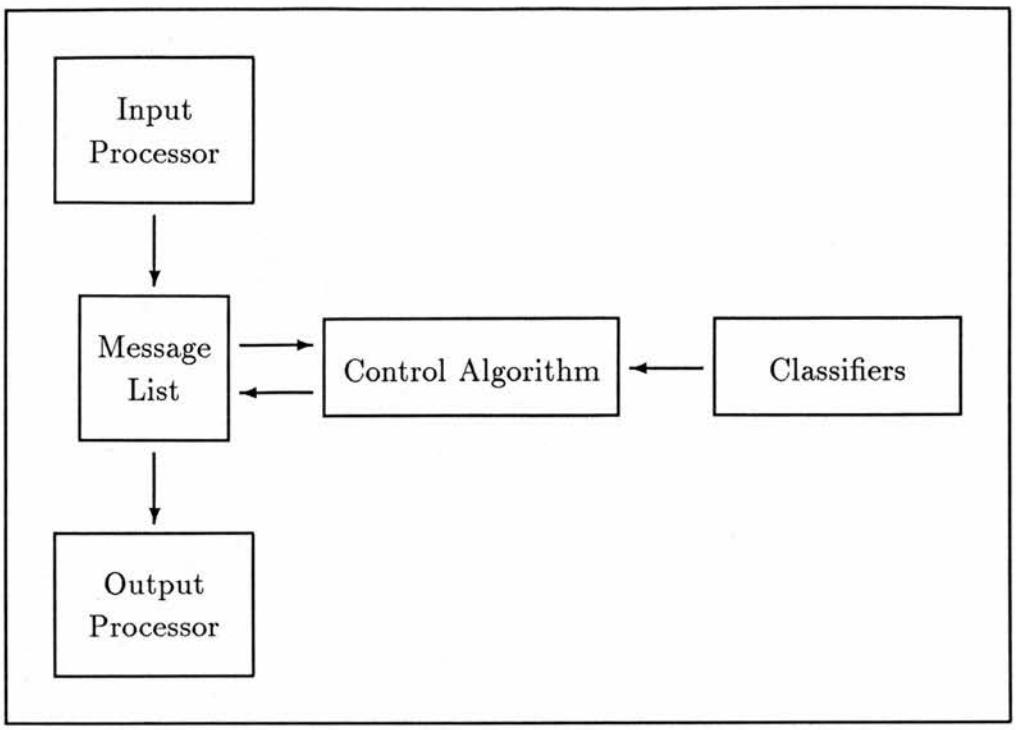

Figure 3–3: A Rule and Message System (RMS) with I/O

message list as that employed by individual classifiers there would be chaos. A controller waiting for a message matching the controller's preconditions would be subject to the posting of a single message, sent by a single classifier, itself a product of random manufacture. While classifiers should tend to improve under the as yet unexplained learning procedures for classifier systems, these learning procedures cannot guarantee that there will not be some minority of classifiers that could greatly reduce the performance of the situated learner, if the messages posted by the vast majority of classifiers are effectively ignored. Either the output processes must examine many messages on the message list, and choose between them, or some control must be placed upon posting messages to the message list in the first place. This is known as the conflict resolution problem, and, in order to solve it, some scheme for assigning relative credence to the various classifiers must be created. To add to this problem, many classifier systems have limits on the number of messages that may be maintained on the message list. In such systems, some method must be used to select amongst competing classifiers.

The most common way used to assign relative credence to classifiers is to base it upon past performance. Often, since the programmer wants an automaton to occupy a particular node of a virtual world, or perform a particular action while occupying a particular node, the automaton is explicitly rewarded when this happens. Because of this, there is a tendency to consider rewards as properties of the world. I therefore want to make a specific point that this is not the case. It is really up to the learning automaton to make the decision on whether it receives a reward for any particular action, and the magnitude of that reward. The world may affect the functioning or even the state of the automaton, but whether this is deemed reward or punishment is a matter for the automaton. Relying on the virtual world to reward the automaton is avoiding the creation of an important subsystem required by the learning automaton if it is to function effectively in the real world. My own research does, in fact, use this expedient, but I think it is important to point out that this is a problem that must, sooner or later, be tackled, and it seems that genetic algorithms may be of use in this regard. Even with the reward conveniently provided by the virtual world, there is still a problem of allocating the reward fairly among all of the classifiers responsible. The classifiers that actually posted the message or messages that caused the last action before the reward was received are obvious candidates to be rewarded, but determining which classifiers posting messages in the past were helpful, harmful, or irrelevant to the goal obtained is a difficult problem, known as the credit assignment problem. As that problem will show up again in a different light, I will leave the issue to be picked up in section 3.5.

Now that the various components (barring the actual learning components) of a classifier system have been described, the sequential algorithm for an RMS with I/O can be stated:

1) Create the initial set of classifiers (This is normally done probabilistically) and the initial message list (normally empty).

2) Add any messages from the input processor to the message list.

3) Select classifiers for execution based upon the current message list.

4) Replace the current message list with the messages produced by execution of the classifiers selected in step 3.

5) Execute output processor. The classifier system may be halted at this point based upon the external world situation, otherwise execution continues at step 2.

Note that this is basically an infinite loop. At some point the system will be halted by external events (*e.g.* the researcher involved halts the experiment, the simulated robot is deemed to have too few resources to continue, or the actual robot hardware deteriorates), but this is not part of the internal algorithm.

Up to this point I have discussed the RMS at a rather high level. This was partially to provide an overview and partially to avoid dissallowing some system from being considered a classifier system merely because of some deviancy from the current standard implementation of an RMS. In fact, current classifier systems have a fairly primitive RMS in order to make them amenable to genetic algorithms[3].

In the archetypical classifier system, each message consists of a fixed length vector, the length of messages being constant for any given classifier system. Each element of a message vector contains a boolean value, which can be thought of as representing the presence or absence of an atomic feature discernible by the system. The taxon of a classifier specifies a conjunction of presence and absence criteria for these atomic features. A taxon need not specify criteria for all of the features detectable by the system, but if any message in the message list matches the detection criteria, the precondition of the classifier has been satisfied. If no message in the message list satisfies the criteria in the taxon, the precondition of the classifier is not satisfied.

---

[3]It is possible that some of this simplification is misguided, since Koza [1990] demonstrates genetic algorithms on Lisp S-expressions and Antonisse [1989] challenges the claim that the number of legal values at a particular site in the genome should be kept small.

Aside from having a taxon, each classifier also has a vector of appropriate size, set with boolean values, to be placed on the message list upon activation of the classifier. Thus, each activated classifier contributes one message to the new message list.

The representations used to represent classifiers and messages within classifier systems are fairly standard. Messages as mentioned earlier, are fixed length boolean vectors. The values "0" and "1" are used as the two possible boolean values, with "0" representing the absence, and "1" representing the presence of some atomic feature. The encoding for classifiers is similar. If the vector length for messages is n, then classifiers are vectors of length $2n$.

The first n elements of a classifier encode the taxon, and are members of the trinary alphabet, {0, 1, #}. In order to determine if a particular taxon is satisfied by a given message, an element by element comparison of the two vectors is made. The taxon is satisfied if and only if each element of the taxon matches the corresponding element of the message being considered. A "0" element in the taxon vector is matched if and only if there is a "0" at the corresponding position in the message vector. A "1" element in the taxon vector is matched if and only if there is a "1" at the corresponding position in the message vector. A "#" indicates that the corresponding element of the message vector is irrelevant to the preconditions required for this classifier, and therefore is matched by either a "1" or a "0" in the message.

The second n elements of a classifier encode the message vector to be posted on the message list when the classifier is activated. These elements are drawn from the binary alphabet previously defined for messages[4].

It may be reasonable at this point to wonder whether these binary messages and

---

[4]Some classifier systems allow a third value, "#". This is interpreted to mean that the message sent upon activation of this classifier is dependent upon the message that satisfied the taxon. In particular, the feature value of the corresponding position in the activating message will be copied into the message posted by the classifier. This is known as a "pass through" position.

trinary taxons can really be expected to effectively encode the sorts of information and condition described when the RMS was first introduced, or even if that is possible, whether an external observer examining the workings of such a system could be expected to be able to decipher what meaning should be ascribed to these bits. While, in general, I think this can be answered in the negative, that is, many of the rules and messages one might originate would be very difficult to effectively encode and it is not necessarily possible to make sense of a system thus encoded, there is a certain amount of order one can enforce upon the system. One way of doing this is to define tags for messages. Messages from the input processor can be tagged such that the first few bits are always set in a particular way, and any encoding may be used for the data portion of such messages. Likewise the output processor may be sensitive only to messages that begin with the first few bits set to some other pattern, and will interpret the data portion in some pre-defined way. With appropriate modifications to the system, one can maintain as much order as one dares.

In the above discussions, I have been talking entirely at the phenotypical level. Even the encoding just described may not affect the genotypical evolution of the classifier system. This is because there are two major approaches to attaching the genetic algorithm to the RMS of a classifier system. One, the "Pitt" approach, uses classifiers as genes, so the actual structure of a classifier is largely unimportant (even though they do commonly use the encoding just described). The other, the "Michigan" approach, uses the classifier as a chromosome, with genetic operations on or between individual classifiers, so the encoding of the classifiers is quite important.

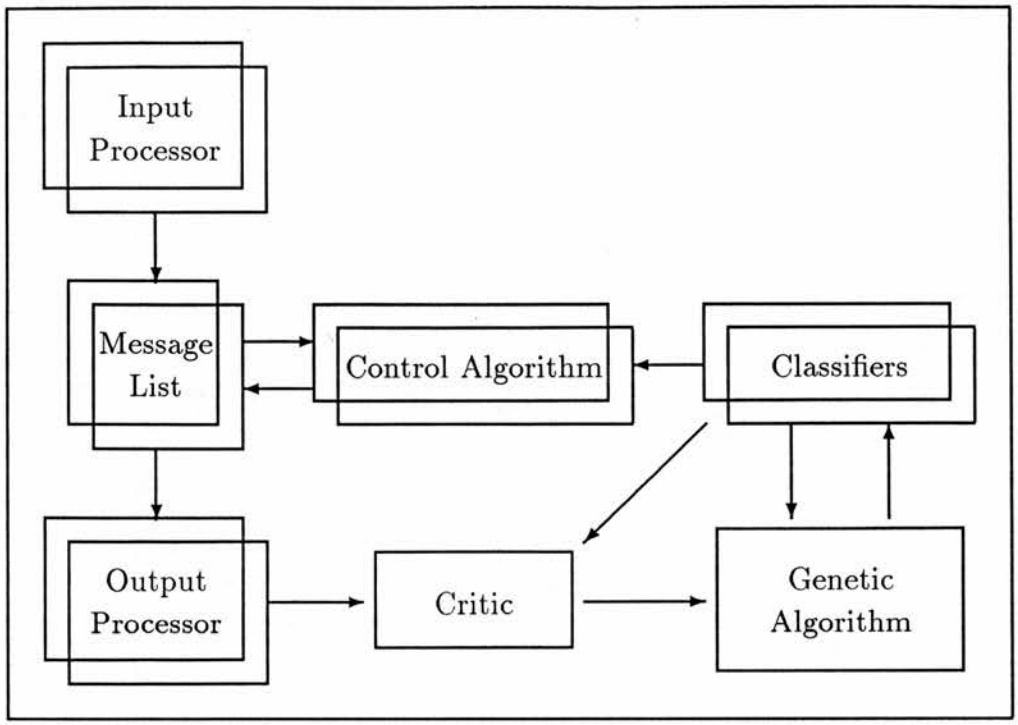

Figure 3–4: A Pitt Style Classifier System

3.3 The "Pitt" Approach

The Pitt approach probably acquired that name because of the research of Stephen Smith [1983] at the University of Pittsburgh in 1980. In the Pitt approach the population consists of many automata such as the ones shown in figure 3–3 with the addition of a "critic" and a genetic algorithm (see figure 3–4). Each RMS (with I/O) is allowed to operate in the problem domain, and its performance is evaluated (via the critic[5]). Higher scoring systems are chosen stochastically for breeding, and a new population created for further testing.

Genotypically, an RMS consists of a sequence of classifiers[6]. Thus, it is classifier sequences that evolve. Classifiers themselves are treated by the genetic algorithm

---

[5]The critic may use genotypical criteria, such as number of classifiers in the RMS, as part of its evaluation criteria

[6]The I/O system, and control algorithm for the RMS do not evolve.

as atomic. They may duplicate themselves, or be forever lost to future classifier systems, but they are not modified[7]. The sequence of operations in a Pitt style classifier system is as follows:

1) Create initial population of rule and message systems.

2) Evaluate each member of the population. This involves doing the following steps until the critic can evaluate the performance of the RMS (perhaps testing each against the others on one or more problem sets):

   2.1) Add any messages from the input processor to the message list.

   2.2) Select classifiers for execution based upon the current message list.

   2.3) Replace the current message list with the messages produced by execution of the classifiers selected in step 2.2.

   2.4) Execute output processor. The classifier system may be halted at this point based upon the external world situation, otherwise execution continues at step 2.1.

   If a good enough member of the population is evaluated, or some other criterion is met (*e.g.* this step has been performed some requisite number of times), then the run is terminated, otherwise, step 3 is performed.

3) A new population is created consisting of allowed genetic operators on members of the previous population, and step 2 is performed (normally some form of crossover and mutation operators are included).

Since the position of a classifier in the genome has no phenotypical effect, one of the rigidities typical in genetic algorithms can be dropped. In the Pitt approach, the size of the genome is allowed to vary between different members of the population. This is accomplished by allowing the points for crossover to be chosen independently for the two genomes. For example (see figure 3–5), if single

---

[7]Actually some mechanism is normally included in these systems in order to create new kinds of classifiers. Otherwise, only a fixed number of classifier types (those that originated in the initial population) would be available.

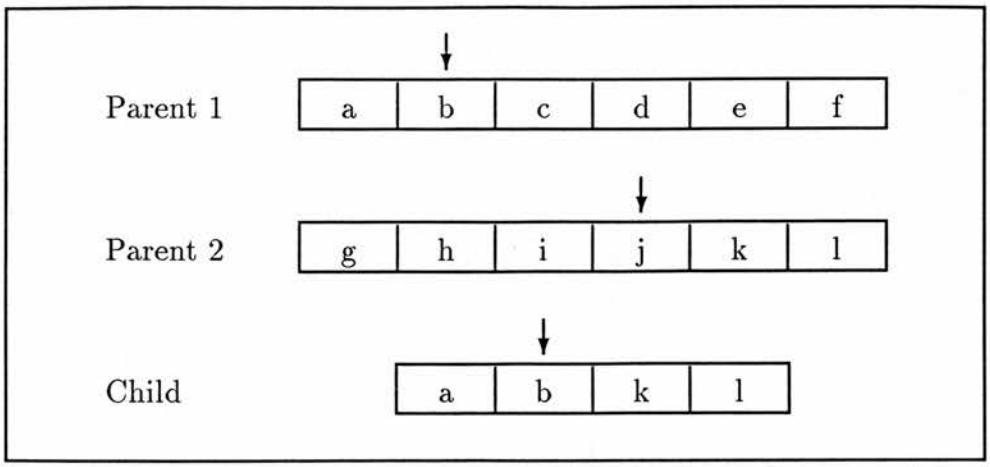

Figure 3–5: Length Mutating Single Point Crossover

point crossover is to be used, a position is chosen at random for the first genome, and a second position is chosen at random for the second genome. An offspring classifier system consisting of all classifiers in the first genome from the start of the genome up to and including the first position chosen, concatenated to all the classifiers from the second genome beyond the second position chosen. Note that this offspring may have a genome vector larger or smaller than either of its parents.

While the Pitt approach has several advantages, it also has several drawbacks. Maintaining a large number of classifier systems, where each can contain a large number of classifiers is expensive. Evaluating the classifier systems can be a very computationally expensive operation. There is no particularly obvious way of generating new classifiers. Most importantly from my point of view, the classifier systems themselves do not learn. I should not overstate these points, as there are many ways around them, and such systems have been very successful. These difficulties did, however, encourage me to adopt a different approach.

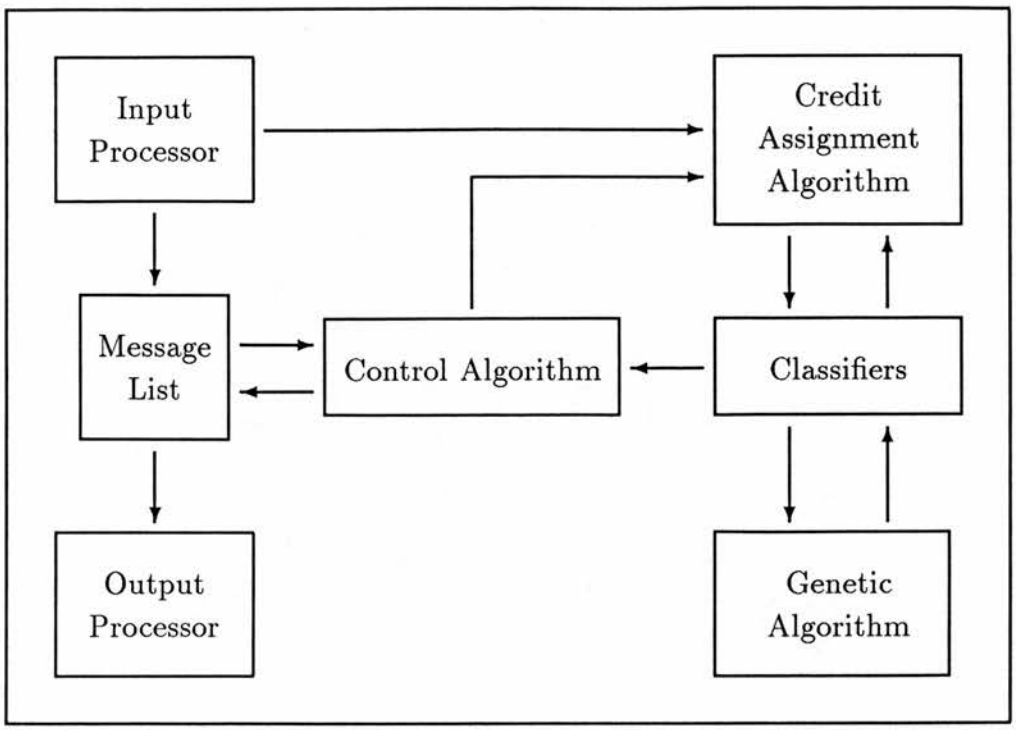

Figure 3–6: A Michigan Style Classifier System

3.4 The "Michigan" Approach

The Michigan approach is so called because it was largely developed by researchers from the University of Michigan. In a Michigan style classifier system, the genetic algorithm operates within a single RMS (see figure 3–6). A fixed size population of classifiers is genetically manipulated. From a phenotypical view, this population of classifiers forms a single classifier system. Rather than the generational genetic algorithm commonly used with genetic algorithms, the Michigan approach only gradually modifies the population of classifiers[8]. Every so often (typically, when the automaton gains a reward) there is a certain probability that a single genetic operation will be performed. A single parent is chosen based upon fitness, or in the case of crossover, two parents are selected. Since the population size is to be kept constant, the offspring classifiers replace existing ones. The

[8]This is similar in its operation to Dewdney's AUTOSOUP described in section 2.2.

classifiers to be replaced are normally chosen probabilistically, either with equal probabilities or in inverse proportion to the fitness assigned to them. The following algorithm describes the basic operation of these systems, although the method for assigning fitness, the "credit assignment algorithm", will be described later[9]:

1) Initialize the rule and message system (create the initial set of classifiers and initial message list). Each classifier in the system is considered a member of the population to be manipulated via the genetic algorithm.

2) The learning automaton is placed within the (probably simulated) environment.

3) Add any messages from the input processor to the message list.

4) Determine classifiers eligible for activation given the current message list.

5) Select some number of eligible classifiers for execution[10].

6) Replace the current message list with the messages produced by execution of the classifiers selected in step 5.

7) The Output processor updates the effectors of the automaton based upon the message list.

8) Some sort of accounting is done in order to assign fitness values to classifiers (*e.g. the bucket brigade*)

9) Some of the population of classifiers is replaced by new ones generated by allowed genetic operators on members of the previous population (some form of crossover is normally included in this set), and step 3 is performed.

---

[9]The algorithm most commonly used with Michigan style classifier systems is the bucket brigade (see section 3.6).

[10]Normally there is a competition between classifiers and only some constant number are chosen for activation probabilistically, based upon fitness. This fitness value is typically computed via some variant on the bucket brigade described in section 3.6.

There are several variants on the algorithm described. Quite often genetic operators of step 9 are only applied after a goal is obtained, whereupon execution is resumed at step 2 rather than step 3.

The Michigan approach has several drawbacks. The genetic algorithm optimizes classifier fitness, not classifier system fitness. Also, it forces us to produce a scheme for evaluating individual classifier fitness. We again face the credit assignment problem.

3.5 Credit Assignment

We are faced now with two credit assignment problems. One is the genotypical evaluation problem which we must always solve in order to use a genetic algorithm. The other is a phenotypical operational problem which must be solved in order to enable conflict resolution within our classifier systems.

The Pitt approach is often seen as avoiding the genotypical credit assignment problem. With rewards kindly handed out by the virtual world, it is normally easy to evaluate the relative performance of competing classifier systems. It is therefore often ignored that individual classifiers are also being rated. In the population of classifier systems, the number of classifiers of a particular type will change from generation to generation. This means that each classifier must implicitly have a fitness. Classifiers are the genes in the Pitt approach. Dawkins argues that in the biological case, it is important to look at the propagation of the genes [Dawkins 1989], rather than just the propagation of the organism. The same appears to be true in genetic algorithms. The Pitt approach does provide a metric for classifier fitness. Whether this metric is better than others is debatable (Westerdale [1989] provides two sides of that debate). In any case, this fitness metric gives us little insight into solving the phenotypical credit assignment problem.

In the Michigan approach, we are immediately faced with the need for a fitness metric amongst classifiers. If we can solve the genotypical credit assignment problem, then we can use the same credit assignment values to solve the con-

flict resolution problem[11]. The most common method in current use for classifier evaluation is known as the bucket brigade.

3.6 The Bucket Brigade

The bucket brigade is an attempt to solve the credit assignment problem by passing back rewards obtained from the environment[12]. It is based upon the concept of an economy[13] where classifiers buy and sell messages, bidding for the ability to do so. Taxes are imposed within the system in an attempt to improve performance.

In order for an economy to exist, there must be a currency. Therefore, each classifier is associated with a quantity generally known as strength. Each classifier is both a producer and a consumer. It produces and consumes messages on the message list. That classifiers produce messages is fairly clear. That they consume them is less so. In order for a classifier to produce a message, an enabling message or messages must exist that satisfy the taxon of the classifier. For the purposes of our economy, the classifier is considered to be a consumer of these messages. A record is kept associating each message on the message list with the classifier which produced it, enabling consumers to pay producers. When a classifier posts a message to the message list, strength (the amount will be discussed later) is taken from this classifier and distributed evenly amongst those classifiers that were responsible for producing the enabling messages (since a taxon may be sensitive to many messages on the message list, there may be several enabling messages, each

---

[11]Whether this is a good idea or not is another question. Especially in the sort of systems I propose in chapter 6, there can be important reasons for maintaining a classifier in the population, yet not desiring that classifier to become active.

[12]As I have stated previously, rewards should not really be a function of the environment, but of the automaton. The use of rewards generated by the environment are an expedient. A robot operating in the real world will not have that luxury.

[13]Holland claims that the bucket brigade is a free market economy, but it is rather different than any free market economy of which I am aware.

one produced by a different classifier). Thus, consumer pays producer, in order to market its own message (becoming itself a producer).

In typical bucket brigade systems, not all enabled classifiers are allowed to produce a message. The message list is limited to some number, m, messages that can be posted. When a classifier has a message to post to the message list (*i.e.* when the taxon of the classifier is satisfied), a bid is generated for the privilege of posting the message. This bid is simply some constant proportion of the classifier's strength. This constant is system wide, so bids are proportional to strength. Of the classifiers bidding, the m highest bidders (some normally distributed noise is added to each bid, just for the purposes of comparison, so that the auction isn't entirely deterministic) are selected. It is this bid (without the noise) that is paid from producer to consumer. A classifier that loses the auction does not post a message and does not have any strength removed.

Aside from the classifiers themselves there is another source of producers and consumers. The sensors of the automaton produce messages. Payment for these messages leave the economy. The effectors controllers "consume" messages. Environmental rewards are paid via the effector controllers to the messages "consumed"[14].

Taxes are used largely to keep useless classifiers from remaining in the system. These are largely of two types. One type continuously pays itself to post its own message (or is part of a ring that performs the equivalent). A percent tax on all income would limit this problem. A second type that may be a problem never post messages. These may be no problem to the immediate operation of the classifier system, but, if created with a significant strength, may be chosen

---

[14]The fixed size message list solves the problem mentioned earlier about single classifiers exerting too much influence on the effector controllers. Since classifiers must compete in order to be placed on the message list, untried or detrimental classifiers will not often place messages on the message list. Because of this, the effector controllers may inspect the message list in much the same way as do classifiers. This means that rewards can be treated as "bids" from the environment, which are divided amongst the classifiers with messages satisfying the effector controllers.

often for replication by the genetic algorithm, and therefore take the place of more useful classifiers. A tax on all classifiers every bidding session tends to remove both of these types of classifier.

There is a certain amount of controversy about the bucket brigade, both philosophically [Westerdale 1985], and pragmatically. There have been a number of attempts to improve or replace it [Huang 1989, Belew and Gherrity 1989]. Indeed, my own research involves severe modification or replacement of the bucket brigade.

3.7 Conclusion

There is a need for some genetically manipulable system that is Turing equivalent. Classifier systems provide a step in that direction[15]. They bring with them two difficult credit assignment problems. There needs to be a way of resolving conflicts between classifier messages determining the actions of the system, and the value of a classifier must be determined for genetic reproduction. The Pitt approach offers one solution to the later problem. The bucket brigade provides a solution to both problems. Other solutions are also possible, and need to be investigated.

---

[15]While I have seen proofs of Turing equivalence for production systems, I do not see how a classifier system without an unbounded message size can have this power. There are only a finite number of possible classifiers, and a finite number of possible messages. Duplicate messages and duplicate classifiers have little effect on the phenotypical operation of the system. This means that for a given message size, there are only a finite number of classifier systems and only a finite number of states in which the classifier system can be placed.

Chapter 4

Rational Reconstructions[1]

4.1 Introduction

Research involves the investigation of new ideas. Part of the investigation is the comparison of a new technique with that of previous works. Most papers specify a problem domain, a proposed solution, and a measurement of the efficacy of the proposed solution. Unfortunately, such papers do not allow ease of comparison between various techniques, because the problem domains differ.

While there is something to be gained by establishing a fixed set of problems, this is not a complete solution. Some techniques may require particular problem spaces in which to excel, and there is always the problem of work that predates the establishment of these standards. These problems are exacerbated in a rapidly developing field such as Genetic Algorithms.

Rational reconstructions are generalizations of existing works [Bundy 1986]. Such rational reconstruction allows comparisons between works, within the common framework of the reconstructed problem and solution domain.

With this motivation, I have attempted a rational reconstruction including the works of Wilson [1985] and Holland [1978]. I chose the first work because it seemed

---

[1]The work reported here is a continuation of that presented at the Third International Conference on Genetic Algorithms [Roberts 1989]. The relevant paper from those proceedings has been included herein as appendix A.

a well documented fairly straightforward design. The latter was chosen because it is the first published work on a classifier system. The work is available in wide distribution, and had been published some time ago, so many people should find the work easily accessible.

Once the results of these works were duplicated, I would be able to test Wilson's work in Holland's problem domain, and Holland's work in Wilson's problem domain. I could test my own systems in these domains, and invent new domains to test all of these systems. Unfortunately, reconstruction is not an easy task. Algorithms are not as well documented as one might think, nor do they always perform in the same way no matter how painstaking the accuracy of the reconstruction. In the scientific community, results that are not repeatable are not considered valid. Particularly in the physical sciences, independent duplication is normally required before a result is accepted. Such independent duplication is known to be a non-trivial task. There are many factors that can influence results. In computer science related disciplines, it is often taken for granted that an algorithm is an algorithm, and that a program on one machine can simply be ported to another. Anybody who has attempted to port a complex piece of software should realize that this is not the case, especially if one has to duplicate the software from specification rather than translating the code.

While it might have been interesting to see how these two reconstructed algorithms would compare on each other's documented problems, my attempts to duplicate each algorithm's performance in its own problem domain varied sufficiently from the published results for these algorithms that such comparison would hardly be convincing. As is often the case in experimentation, my results both confirmed and conflicted with the published accounts. Since I have not imported the exact programs used in the published works, this thesis in no way attempts to repudiate these works. In fact, the confirmation of comparative results without duplicating the actual results can be seen to strengthen the claims made in the articles. Still, the differences serve as warning that published articles rarely contain sufficient information for duplication, and trivial variations can produce considerable differences in results.

4.2 The Classifier Systems

Both Wilson's and Holland's systems work in deterministic finite state worlds, consisting of nodes and edges, where all the edges between nodes are bidirectional and the connectivity pattern of nodes is regular. The learning automaton is placed in one of the nodes. An action by the automaton is a request to move along one of the edges of the node it occupies to another. The automaton continues to produce move requests until it arrives at a node that is marked as a goal node. At this point, the reward handling mechanisms are invoked and the simulation is restarted.

Both algorithms are basically "Michigan" style classifier systems (*i.e.* the genetic algorithm works on classifiers within, rather than between, classifier systems[2]). The genetic algorithm is used to generate new classifiers, based on the performance ratings of the old classifiers. There is variation in both Wilson's and Holland's automata from the description given in chapter 3. These differences will be characterized in the relevant sections (4.3 for the Animat, and 4.4 for CS-1). Within the "Michigan" framework, however, both systems use goal completion as a trigger for genetic manipulation (*i.e.* genetic manipulation is performed only when the automaton reaches a goal node).

4.3 Wilson's Animat[3]

Wilson's classifier system is the simpler of the two systems I attempted to reconstruct. Despite this, it is capable of operating in more topologically complex

---

[2]Westerdale offers an analysis that would put Holland's work in the "Pitt" framework, because it is a global reward scheme [Westerdale 1989]. It remains to be seen what distinctions will be considered salient, and I have used the historical convention.

[3]Further consultations with Wilson did not significantly change the results presented here.

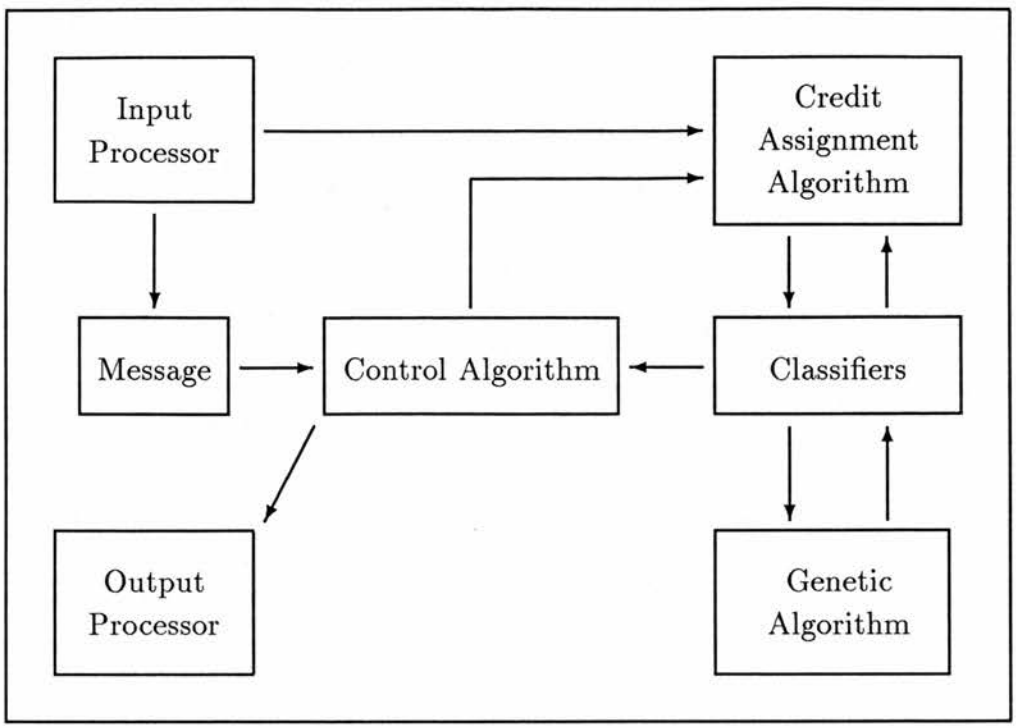

Figure 4–1: The Animat Classifier System

environments. Globally the world is topologically toroidal, and the local connectivity is eightfold. Each node is describable by two bits (which Wilson describes as "food smell" and "opacity"). As the automaton is presumed to be sensitive to the nodes surrounding it, each node is labelled with the bits (starting from North and preceding clockwise to form a sixteen bit string) descriptive of the eight nodes surrounding it. Wilson named the world described in his paper "WOODS7"[4].

While the system is similar to the Michigan system described in section 3.4, there are significant differences (see figure 4–1). In particular, the message list has been pruned down to handle only a single message (produced by the input processor), and classifiers compete to send "actions" to the output processor (via the control algorithm)[5]. Also, The two credit assignment problems detailed in section 3.5 are not handled identically. That is, the phenotypical rating, used to

---

[4]For a further description of the world used in Wilson's simulation, see section 5.3.

[5]While this could be described in terms of a length two message list with various

determine whether a classifier will have its action selected by the control algorithm, differs from the genotypical fitness used by the genetic algorithm to determine whether the classifier will be chosen for a genetic operation.

The use of actions rather than messages gives rise to the concept of an action set. Those classifiers that match a situation and specify the same action are referred to as an action set. When the Animat selects an action, those classifiers that match the current situation and specify this action are the current action set. For the purposes of the accounting algorithm, all members of the current action set are treated the same.

The difference between the phenotypical and genotypical credit assignment algorithms is due to a new property associated with each classifier, a *distance* value, which is an estimate of the number of actions that will be required in order to reach a goal from the point where this classifier is satisfied. This information is combined with the standard *strength* value in the phenotypical classifier rating scheme.

Since this is a concrete example of a classifier system, some of the areas left ambiguous in section 3.4 can be clarified (in particular, the credit assignment algorithm), and some special case handling (reward situations) appears in the algorithm for the Animat:

1) Create initial set of classifiers. The current action set is empty.

2) The Animat is placed in the virtual world with an empty message list.

3) The input processor adds a message to the message list based upon its sensory equipment.

4) Determine classifiers eligible for activation given the current message list.

5) Determine relative priorities of eligible classifiers based upon *strength* and *distance* information.

---

restrictions, thus fitting better into the model created in chapter 3, I do not believe that it is particularly enlightening to do so.

6) Choose probabilistically amongst eligible classifiers. A new current action set, consisting of all eligible classifiers that specify the same action as the classifier chosen, is formed. The previous "current action set" is still of interest, and is referred to as the previous action set.

7) Accounting is done for all of the classifiers in the system.

   7.1) *Strength* is taken from members of the previous action set, and is distributed evenly between members of the current action set.

   7.2) A distance estimate is taken via examination of the current action set, and this is used to update the *distance* values of the previous action set.

   7.3) All classifiers are taxed a percentage of their *strength* values.

8) The output processor updates the effectors of the automaton based upon the action selected.

9) Reward information is made available to the automaton (*i.e.* whether a goal node has been reached and, if so, the magnitude of the reward).

10) If a goal node has been reached, then:

   10.1) The reward is distributed evenly between members of the current action set.

   10.2) The members of the current action set have been shown to be a single action away from a goal. The *distance* values of these classifiers are updated in order to reflect this fact.

   10.3) Possibly perform a genetic operation (recombination, duplication, intersection and two forms of mutation[6] are supported).

   10.4) Go to step 2

   Otherwise, go to step 3.

---

[6]Both of these mutation operators differ somewhat from the one used similarly in later systems I devised (see section 4.5).

Before going into the details of this algorithm, a description of the the main data structures may be helpful. The node labels of the virtual world are used directly as the message placed by the input processor onto the message list. Thus messages are sixteen bit binary strings. The classifiers consist of a sixteen position vector, each position drawn from the set {0, 1, #}, which is the taxon of the classifier[7], and an action specifier (which takes the place of the message in conventional classifier systems), taking the value of one of the eight possible directions of motion. Also associated with each classifier is certain accounting information, a *strength* and a *distance* value. The *strength* is a measurement of the utility of a classifier. The *distance* is an estimate of the number of actions (moves) that are required to reach the nearest goal at the time that the classifier matches the current node.

Ignoring step 1 for the moment, we will start discussing the algorithm at step 2. The Animat is placed randomly in some unoccupied location in WOODS7. In the following steps of the algorithm, the Animat determines its action. The automaton receives the information about the node (contained in the node label). The set of classifiers with matching taxons is determined[8]. Of these, each receives a rating ($rating = strength/distance$), and one is selected (with probability, $rating/\sum ratings$). The action specified by this classifier is the action sent to the output processor[9]. *Strength* and *distance* values must then be updated.

A form of the bucket brigade is used to update *strength* values. All classifiers (that match the input) that specify the action selected (now the current action set) are treated identically: they have a fraction (0.2) of their strength removed, and

---

[7]As is standard in other classifier systems, a "#" in the taxon means that the feature at that locus is not salient to satisfying the taxon.

[8]If none match, or if the composite strength of the matching classifiers is below some threshold (100), a matching classifier is engineered on the spot, either at random, or using a rather *ad hoc* technique which I will discuss later.

[9]There are obscure cases where $\sum ratings$ is zero. In these cases, an action is chosen at random.

the strength thus obtained (remember that in the bucket brigade, classifiers must pay in order to become active, and the currency is strength) is divided equally amongst all the classifiers that were members of the previous action set[10]. Thus, reward information is passed back to classifiers that set up the current situation. Likewise, distance information must be passed back to earlier classifiers.

In order to get an estimate of the automaton's current distance from a goal, the mean is taken over all *distance* values of classifiers in the action set. This value is used to update the *distance* values of the classifiers in the previous action set. The *distance* value for a classifier is actually a running average of three values stored along with each classifier. The running average of all classifiers in the previous action set is updated to include the distance estimate of the current action set. Since the classifiers of the last action set are clearly one step farther from a goal than the current action set, this fact is included in the update (*i.e.* a new value is placed in the running average of each classifier in the last action set that is equal to the mean distance estimate of the current action set plus one).

Step 7.3 removes a fraction (0.05) of the *strength* from every classifier in the system. This keeps classifiers that are never used from maintaining high *strength* values. This could otherwise be a problem since the genetic algorithm would propagate these useless classifiers, destroying classifiers that are of more utility.

Each time a goal is reached is considered one "iteration" of the system. When this happens, many things must be done. *Strength* and *distance* values for the current action set must then be updated. The reward (uniformly 1000 in Wilson's reported research) is divided equally amongst the members of the current action set. The distance from the node that satisfied the current action set to a goal node is known to be traversable with a single action. The running average of all classifiers in the current action set are therefore updated with unity. At this point,

---

[10]In the obscure cases where $\sum$ *ratings* is zero, where an action has been chosen at random, the previous action set is rewarded as if there were a single classifier in the current action set with strength equal to the average strength of all classifiers in the system.

a genetic operation may be invoked. Each time a goal is reached is considered one "iteration" of the system. Four genetic operations are supported: two point crossover, intersection, duplication, and two forms of mutation operators[11]. Note that genetic operators do not destroy the parents, but the population of classifiers is kept constant, so some classifier is destroyed for each genetic operation performed. The scheme for selecting classifiers, both as parents and for deletion, will be discussed after all of the genetic operators have been described.

Two point crossover has been described in chapter 2. The only thing that needs to be discussed is the *strength* and *distance* values. There is some sense in which *strength* is conserved in these operations. One third the *strength* is taken from each parent and given to the child[12]. The *distance* value for the child is just the average of the *distance* value of each parent[13].

Duplication is merely creating an additional copy of an extant classifier. The *strength* of the parent is divided evenly between parent and child, and the entire *distance* information copied. This has no immediate effect on the functioning of the automaton, but helps preserve the genotype of the classifier being duplicated in the face of later deletions.

Intersection is an interesting operator invented by Wilson that is peculiar to classifier systems. Intersection is a technique for creating generalizations within a classifier system. As with two point crossover, two cut points are selected. At ev-

---

[11]These operators are called "create" operators. They do not actually depend at all upon the genetic makeup of the population, but are more in the spirit of spontaneous generation. More will be said about them later.

[12]Since the child replaces some classifier that will likely have a non-zero *strength*, the actual classifier system loses that *strength* so the average strength of the system is reduced by some amount.

[13]While *distance* is normally a running average of three values, sometimes (*i.e.* whenever a new classifier is created other than by the duplication operator), only a single value is stored (It is not clear what Wilson does in these cases, but it appears unlikely to be significant).

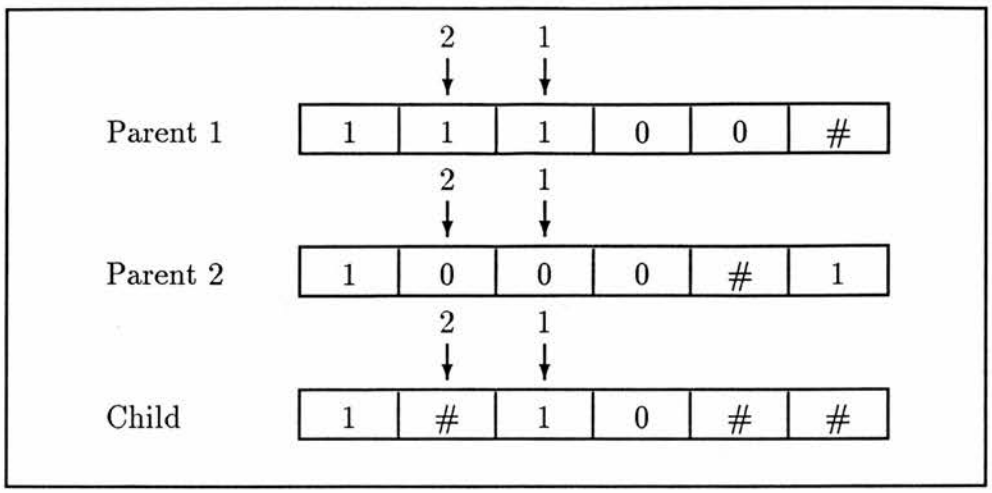

Figure 4–2: Intersection (linear representation)

ery position where the elements from the second classifier would normally appear in the child[14], instead the following algorithm is used: If the two parent vectors are lexically identical at that position, the child inherits that allele. Otherwise, the child inherits a "#" at that location. Thus, the taxon of the child is a generalization of the taxon of the first parent (*e.g.* as shown in figure 4–2). The *strength* and *distance* values are computed as for crossover.

We have already encountered the mutation operators. They are exactly the operators used when no classifier matches the input message. The first is a "random create". With no parent required, a new classifier is added to the system. For each locus on the taxon, the value from the current node label is used with two thirds probability, and one third probability of being set to the value "#". The action specifier is chosen uniformly at random from those available. The *strength* is set to the average *strength* of all classifiers in the system, and the *distance* set to the average of all the *distance* subvalues[15] of all classifiers in the system.

---

[14]That is, from just beyond the position chosen as the first cut point up to and including the position chosen as the second cut point.

[15]By "subvalue" I mean the actual components from which the *distance* value is

The second mutation operator is a non-random "create". It involves moving from the current position and examining spots near by. I object to this operator on the grounds that an actual robot would not have the luxury of testing what would happen if a particular action were taken without taking such an action. I will have more to say on this subject later, but for now I will discuss how I implemented this operator[16].

In my implementation, all eight nodes surrounding the current position were investigated. For each node, classifiers with taxon's matching the label of the node were gathered in order to estimate the distance of that node from a goal. From the classifiers that match a node, a weighted average is taken, the *distance* value of each classifier weighted by its *strength*. If the best node investigated has an estimated distance that is one or more less than the estimate for the current node (estimated in the same way), then a new classifier is created, the taxon constructed in the same fashion as for a "random create", but with the action chosen to propel the automaton in the direction of best node investigated. The *strength* and *distance* values are set as for a "random create".

Upon reaching a goal, a genetic operator may be invoked, but this is not necessarily done. There is a 0.25 chance that one of crossover, duplication, or intersection will occur, and an independent 0.02 chance that a "create" operation will occur. The distribution in case one of the first three genetic operations was chosen was: 0.25 for crossover, 0.5 for duplication, and 0.25 for intersection[17]. Random and non-random creates are equally likely (*i.e.* probability 0.5).

---

composed. Thus, a newly created classifier might have less effect on the average than one that has been activated many times.

[16]This is indeed one of the areas where my reconstruction may differ from Wilson's Animat, and that will be discussed.

[17]This conflicts with the numbers in Wilson's Animat publication, because intersection was not considered in those figures. Private communication from Wilson dated 16 May 1988 confirms the probability distribution used in the reconstruction.

When classifiers were required for a genetic operation (as with crossover, duplication, and intersection) they were chosen with probability proportional to *strength* (*distance* is not considered). If the operation is crossover or intersection, a second classifier is chosen from those with the same action (again with probability proportional to strength). Since the population of classifiers is to remain of constant size, whenever any genetic operator is invoked, a classifier is selected for deletion (with probability inversely proportional to strength).

This leaves only step 1 from our algorithm still to explain. The only part of the classifier system that needs initialization is the actual data base of classifiers. The initial set of 400 classifiers is generated probabilistically. The taxon for each classifier is constructed as follows: for every position in the taxon vector, a value of "0" is chosen with probability 0.25, a value of "1" is chosen also with probability 0.25, and a value of "#" is chosen with probability 0.5. The action for each classifier is chosen from a flat distribution over the set of actions available to the system. The *strength* of each classifier is initialized to 100, and the *distance* is set to unity[18]. From this knowledge poor start, the system learns to gain rewards from the environment.

Wilson's system achieved a goal in an average of four or five steps after training. This was compared to an optimum of 2.2 steps average for an omniscient automaton. As Wilson's Animat has a very limited sensory range, and cannot explicitly store internal state information, this optimum seems unobtainably low. To put things in a more realistic perspective, I designed a set of classifiers and disabled any mechanism that would change this set. No new classifiers would be created by the system and no old ones deleted. The rating of each classifier was fixed for the entire run. The result was a system that achieved a goal in 3.1 steps on average. While this may, or may not, be the optimum obtainable for a single state automaton with the sensors available, it at least forms an upper bound. It also

---

[18]The *distance* is actually an average value of a list of up to three past distance estimates. Upon initialization, the list consists of a single element, one.

serves as a reasonable goal for a learning automaton with the same restrictions. Wilson's Animat results are quite respectable.

Unfortunately, I was unable to reproduce the documented performance of Wilson's Animat with my own reconstruction. After nine thousand iterations, the average performance of the reconstructed automaton over the next thousand iterations averaged 6.65 steps over 41 individual runs[19]. These results imply a mean significantly ($< .005$ chance that this is a random effect) greater than 5. Wilson's Animat performs significantly better than my reproduction.

There is another interesting potential difference between the results. As in most "Michigan" style classifier systems, there are two learning algorithms in Wilson's Animat. One form of learning is performed by the genetic algorithm (producing better classifiers), and the other form of learning is performed by the bucket brigade (evaluating the classifiers). Wilson reports no significant degradation of performance when the genetic algorithm is removed. Turning off the genetic algorithm in my own routines produced an average of 11.4 steps to solution in comparable circumstances to the results reported above. This was the average over nine separate runs. The standard deviation of this mean over individual runs was 2.6. This is significantly ($< .005$ chance that this is a random effect) worse than the performance with the genetic algorithm.

4.4 Holland and Reitman's CS-1

Holland and Reitman [1978] describe a complex automaton, with multiple sets of classifier systems. A much simpler subset is actually implemented[20]. Because of this, only two actions were available to the system. Given this restriction, and

---

[19]While this data is from an entirely new set of runs from those previously reported [Roberts 1989], incorporating several rather minor changes, there is no significant difference between this sample mean, and the mean of 6.5 (with a standard deviation of 0.6 over 20 runs) previously reported.

[20]by Wright and Forman

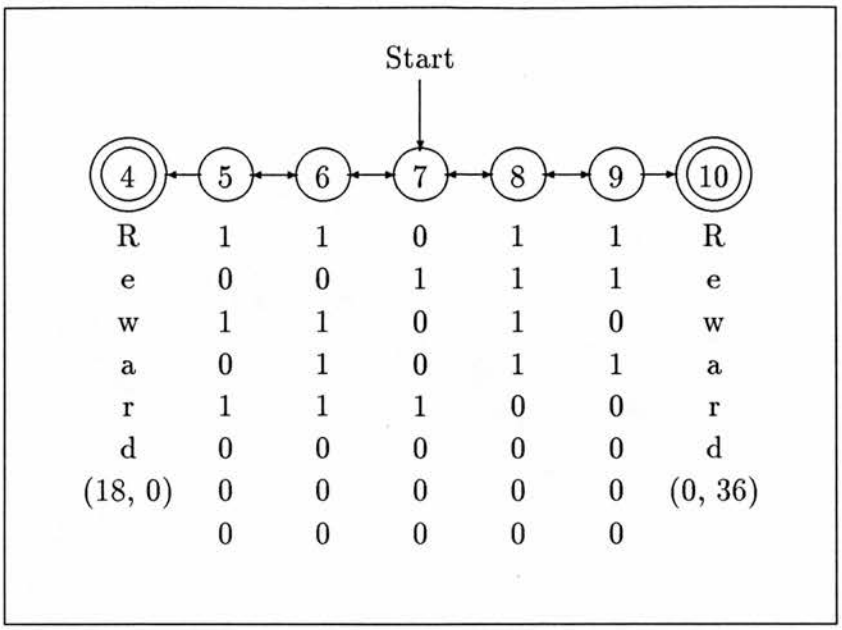

Figure 4–3: Holland and Reitman's Environment A

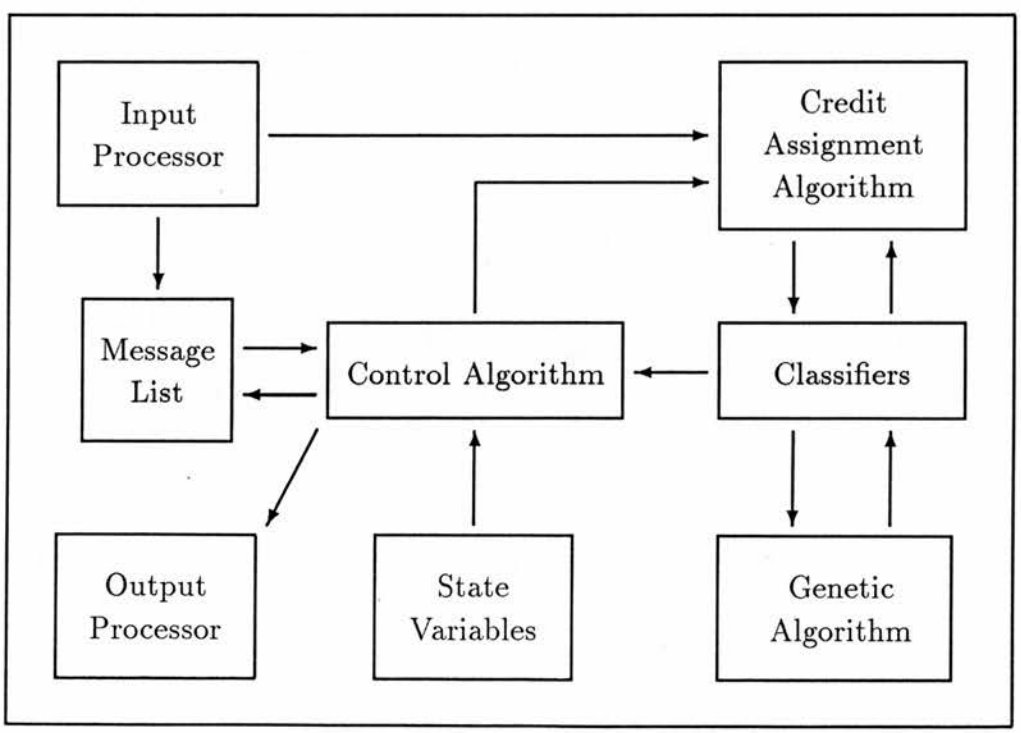

Figure 4–4: CS-1

the bidirectional nature of edges between nodes of the world, only two classes of topology are possible, either a line segment, or a ring. Since the simulation halts when a goal node is achieved, all worlds simulated are effectively a sequence of nodes in linear progression where the end nodes are goal nodes. The virtual world shown in figure 4–3 was one of two used by Holland and Reitman, called world A (since reaching a goal node immediately halts the simulation, edges to goal nodes are not shown as bidirectional in the figure). This is the virtual world chosen for my reconstruction of their work.

There are several places where the work is vague. In such cases, I have been forced to guess at the methods of the original implementation, and I include a description of what I have implemented. As my results differ markedly from the published work of Holland and Reitman, presumably not all of these interpretations were accurate[21].

I have included this section in order to show some of the difficulties encountered in the reconstruction effort, how guesswork can still provide interesting results, and as another illustration of an actual classifier system. I have not included painstaking detail of the actual program I implemented, because that is not the point of this section. I do not recommend attempting to duplicate my reconstruction of CS-1, nor do I recommend attempting a reconstruction of the CS-1 described by Holland and Reitman [1978] with only that text as a guide.

In Holland's design the architecture of the classifier system (see figure 4–4) is rather more complex than in Wilson's (or in the typical classifier system). The system has internal resources affected by external events. In the simulation, two internal resource reservoirs were used. The classifiers are more complex as well.

---

[21] Since implementing my reconstruction, I have obtained an unpublished document by Wright [*circa* 1977] that elucidates many of the uncertainties that arose during my reconstruction effort. The document describes a preliminary version of CS-1. As this version is clearly different than that presented by Holland and Reitman (and produced "quite disappointing" results), I have not integrated the information contained in Wright's document into the text of this section. My guesses seldom matched the procedures documented by Wright.

They are almost like a composition of Wilson's taxon and the typical classifier system taxon, plus a few other items as well. The taxon is not only sensitive to the environment (as is Wilson's), but also to a message list (as in typical classifier systems) and the last action of the automaton. Holland's classifiers have an action (like Wilson's but consisting of a single bit), for controlling external actions, and a message bit string, to put on the message list. The accounting section consisted of four items: predicted payoff[22], age, frequency, and attenuation. Predicted payoff is an estimate of the resources available at the goal that will be reached if the classifier is activated. Age is an indication of how long the classifier has existed (although the age of a classifier actually decreases when it is used). Frequency is a simple count of how many times the classifier has been activated. Attenuation is an estimate of how culpable the classifier is to be considered in eventually reaching the goal.

The general algorithm for CS-1 is similar to that of Wilson's Animat. CS-1 is placed in the simulated world where its rule and message system are allowed to function until a goal node is reached. The exact functioning of the algorithm is rather different. The functioning of the rule and message system (along with I/O) is known as the stimulus-response cycle. In the stimulus-response cycle, each classifier that matches the current situation is given a priority based on the number of non-wildcarded fields multiplied by "the amount of the current needs fulfilled by this classifier's predicted payoff"[23]. One of the top ten classifiers is then selected probabilistically, based upon its relative priority. The selected classifier has its action performed, its age halved and its frequency incremented by one. If the predicted payoff of the selected classifier is less than that of the previously chosen classifier, the attenuation factor of each classifier chosen since the most recent goal achievement (excluding the currently selected classifier) is incremented.

---

[22]This is not a single value. One value is kept per internal resource reservoir. So each classifier had two predicted payoff values per classifier in the reported simulation.

[23]The precise meaning of this quote is not obvious. I give my own interpretation later in the text.

The automaton learns only when a goal node is reached. First, a non-genetic learning scheme is applied. The classifiers that have been activated since the previous learning cycle have their predicted payoffs modified to reflect the achieved goal. "Those predicted payoffs that were consistent with (not greater than) this reward are maintained or increased; those that overpredicted are significantly reduced, all according to their attenuations." All attenuation factors are reset to zero.

The creation of new classifiers is accomplished by the crossover operator. The parents are selected each with "probability proportional to its predicted payoff". The new classifier has predicted payoff and action taken from one parent, and an age that is the average of the two parents. Attenuation factors and frequency are set to zero. In order to make room for the new classifier, an old classifier is deleted. From the classifiers with the greatest age, the one with a taxon most closely matching the new classifier is chosen for deletion.

In Holland's work, the use of genetic learning significantly increased the learning speed (and possibly the asymptotic results as well) of the learning automaton, both in an initial learning case, and in the case of an experienced automaton in a new environment.

In reconstructing Holland's work, several questions must be answered. There are still a number of important ones outstanding, as well as a problematic case that I expect needs to be solved. One must determine the "current needs fulfilled" given a payoff vector. A comparison function for payoff vectors is needed. A mapping from payoff vectors to scalars must be made. A function must be designed to modify classifier payoff vectors, "all according to their attenuations". The problematic case is that according to the algorithm described, a classifier might not be strongly affected by a particular payoff, even if that classifier was the last to be used to obtain the payoff. The reason is that it might have been used earlier, and thus have a very high attenuation value.

In order to create a "current needs fulfilled" value for my reconstruction, the automaton is considered to have a needs vector (similar to, but opposite in magnitude from the original article's resource reservoirs). Each element of the vector was incremented after an action was performed by the automaton. The "current

needs fulfilled" is derived by summing the minima of corresponding positions in the needs and payoff vectors. When a goal node is achieved the reward for obtaining a goal is subtracted from the corresponding need. If the result is less than zero, the need is set to zero.

The comparison function on payoff vectors, for purposes of determining attenuation factors, is a simple comparison between each element in the payoff vectors. If any value in the currently selected classifiers payoff vector is less than the corresponding element in the previously selected classifier's payoff vector, the attenuation factors of all previously selected classifiers (since the last achievement of a goal state) have their attenuation factors increased. Note that this comparison function is not a partial ordering.

A mapping of payoff vectors to scalars is needed in order to select parents, for new classifier construction, with "probability proportional to their predicted payoff". The function chosen was simple summation of the payoff vector.

Modification of payoff vectors was accomplished in the following manner. For each element in the payoff vector, a weighted average is computed ($average = ((frequency - 1) \times payoff + reward)/frequency$). If this average is greater than the payoff, then the payoff is increased ($payoff_{t+1} = ((attenuation + 1) \times payoff_t + average)/(attenuation + 1)$), otherwise it is decreased ($payoff_{t+1} = average$).

As stated earlier, there is a problem with handling classifiers that are used multiple times in achieving a goal, since the attenuation factors do not appear to be reasonably handled in the scheme outlined by Holland. Because of this, the results given here are based on a scheme where attenuation factors are kept for each selection of a classifier. Payoff vectors were modified as above, with the most recently selected classifier updated first.

The criteria for success used by Holland is a bit more involved than that of Wilson, due to the needs of the automaton being vector defined. Since achieving a goal might only satisfy one of several needs, it is not enough to give the average number of steps to a goal. Holland solves this by giving a specific set of conditions for success, based upon the actual nature of the world simulated. There are two goal nodes, each providing a different one of two needs. One goal provides at twice the magnitude of the other. Holland's criteria are ten consecutive trials, where the

minimum possible steps to a goal is taken, and one goal is visited twice as often[24] as the other (since both resources are depleted at a constant rate, I presume the goal providing the least resource should be visited most frequently[25]). The better the learning algorithm, the fewer iterations before this occurs.

For Holland's environment A, Holland's automaton achieved the success criteria after 212 iterations. Attempted without the genetic algorithm, it took 2161 iterations. My results, in the same environment led to the following observation. As the reward at each goal node was amply sufficient to satiate the needs of the automaton, there was no preference to move to one goal rather than another. Random variation eventually produced ten consecutive optimal trials with the ratio six to four (for ten trials, this is roughly two to one) after 6363 iterations, while the first ten consecutive optimum trials to goal (in a one to one ratio) occurred after 839 iterations. This optimum performance was repeated 19 times (once in a four to six ratio, all other in a one to one ratio) before Holland's criteria were met[26].

---

[24]Note that the ratio of seven to three is the closest one could get to a two to one ratio with ten trials.

[25]This reasoning agrees with that in Wright's document.

[26]Apparently, my reconstruction does far better than Wright's preliminary version. Wright characterizes the performance of his system as "only slightly better than the predictions made by the random walk model". If we used a coin toss to determine the direction to move, the chance of moving directly to the left goal (three moves left) from the start is $0.5 \times 0.5 \times 0.5 = 0.125$, and the chance of moving directly to the right goal is the same. Thus, the chance for moving directly to a goal is $0.125 + 0.125 = 0.25$. The chance of doing this ten times consecutively is $0.25^{10} = 4^{-10} = 1/1048576$. The reconstructed CS-1 does this much more quickly than the 1048576 iterations predicted by the random model. Without the genetic algorithm my reconstructed automaton did not get ten consecutive optimal trials even after ten thousand iterations. Due to a third action (move the same way as last time) available according to Wright's document, he may have estimated a random walk model that would accomplish ten optimal trials in an average of just over 3325 iterations.

4.5 Conclusions

The difficulty of comparing various published techniques remains formidable. Even the best documented algorithms can defy duplication. It remains for researchers to do the best they can in providing all of the salient information. Making the code available as well is also useful, although differences in programming languages tend to limit ease of transport.

So much guess work was involved in the reconstruction of Holland and Reitman's CS-1, that it is hardly surprising the attempt was unsuccessful. It is clear that many differences exist between my reconstruction and the original. I think little will be gained by my further guesses as to which differences are the significant ones. The reconstruction of Wilson's Animat is more perplexing.

Only a bit of guess work was required in the reconstruction of Wilson's Animat. Wilson himself could be consulted for answers to various questions. Fortuitously, many of the choices I had guessed at turned out to be the same as Wilson's implementation. For example, reward values, initial *strength* and *distance* values for classifiers, even the choice of two decimal precision, fixed point arithmetic turned out to coincide with Wilson's implementation. There is no question, however, that significantly different results were obtained, which would imply significant differences between the reconstruction and the original. It is of interest, therefore, to attempt to identify some of the remaining potentially significant differences.

One promising area for significant differences is that of "creates". These are the operations that spontaneously generate classifiers as execution proceeds. Wilson identified these as important in obtaining the performance he achieved. There are two forms of the create operator. One form generates classifiers fairly randomly, but with a taxon based upon the current location. The other is a kind of look ahead operator: it performs distance estimates on adjacent positions, and creates a classifier specifying movement in that direction if such a classifier is deemed advantageous, based upon the distance estimates. Note that this operator is something like having the ability, when faced with a question of which of two doors to use, of stepping through each of the doors in turn, and then creating a rule to go to

the best of the doors investigated. The treasure will be obtained and the hungry lion avoided, all without risk. While I have noted much ambiguity about the exact way in which this feature was originally implemented, I have investigated several variants of this operator, none of which significantly improved performance. I have also experimented with changes that affect both types of creates, and several of these deserve further comment.

There is something a bit strange about having either type of create operator in step 10.3 of the algorithm for the reconstructed Animat. When these mutations are applied, the Animat has just proposed an action that will lead it to a goal state. This is a strange place to want to generate a classifier that would suggest moving in a random direction, or even a non-random direction. There is already a classifier in the system that applies to this state, and suggests moving on to the goal state, otherwise the automaton would not be at this point in the algorithm. Because of this, I have tried variants that avoid this peculiarity. Two of these schemes involve generating the taxon of the new classifier from some other source than the current message. The third moves creates to a different part of the algorithm.

The first of these schemes is simply to generate each position in the new taxon probabilistically[27]. While use of this scheme did not noticeably improve the performance of the reconstructed Animat, it was adopted for use in the lookahead systems described in chapter 6.

Another scheme for generating the taxon of the new classifier was to use some random spot in the virtual world as the source. While this seems objectional to me for roughly the same reasons as I object to non-random creates, it did pro-

---

[27]The precise relative probabilities for "0", "1", and "#" are 3, 1, and 4, respectively. The relative probabilities are *ad hoc*, representing my own impression of what might be useful, given the problem domains used. A better approach might have been to sample the relative frequencies encountered in the past for each position, and use that to choose the relative likelihood for "0", or "1", but that still leaves an *ad hoc* choice for "#".

duce a significant improvement to the performance of the Animat[28]. The scheme still produced significantly worse results than Wilson's original work, but they do provide some promise that Wilson's result can indeed be reached by more such experimentation.

Finally, I have reinterpreted a personal communication by Wilson dated 10 August 1988. It leads me to believe that the following changes would more closely duplicate Wilson's work. No longer perform create operations when rewards are obtained. In the other place the create operator is invoked (when there are either no classifiers matching the current message, or these classifiers are below the strength threshold) make the following changes. If the operator is not invoked, invoke a random create with probability 0.02. If still no create operation has been performed, invoke a non-random create with probability 0.02. This increases the use of creates by at least a factor of eight from that reported here, and performs them at more sensible times. Unfortunately, preliminary results with these changes have not been promising.

I am still confident, however, that some variant of the above techniques may yet reproduce the results obtained by Wilson. If, as I suspect, something about creates is indeed the reason for the difference between the results reported for Wilson's Animat and my reconstruction, it is not particularly important for the main purpose to which I have used the reconstruction, which is comparison of lookahead versus non-lookahead systems. I believe I have discredited the non-random create sufficiently that, aside from being some inspiration to do true lookahead, it needs no further consideration. The random create, however, while not being entirely genetic, is still a reasonable concept for machine learning. My argument here is two-fold. First, the create operator is, in a sense, orthogonal to my research. That is, the systems I have proposed can (and to a certain extent, do) make use of

---

[28]The probability with which both types of creates occur in step 10.3 was doubled for these runs.

this operator. Second, the relative results determined empirically would not be changed[29].

On the whole the results obtained from my reconstructions confirm the utility of the genetic algorithm in solving problems within the domain specified. The differences in detail, however, are quite marked. Because of this, the attempt to compare Wilson's Animat with CS-1 was not carried through. The reconstruction of Wilson's Animat was kept for comparison with later classifier systems I devised, but CS-1 was dropped from further study.

---

[29]Improved performance on the part of the reconstructed Animat in Octworld or Hexworld would hardly counter the evidence already obtained, and all create operatives were incapacitated in N-World (see chapter 7).

Chapter 5

The Problem Domain

5.1 Introduction

Ideally, we would like to solve the problem of autonomous goal achievement in an environment as complex as the universe, starting with little or no knowledge of the universe. This is not to say that the system would initially be able to achieve any given goal, but that, should it manage to avoid destruction, it becomes better at achieving goals as it gains experience. The universe being a very difficult thing to formalize, a simpler model is described.

In order to leave things as open as possible, the universe is described only in terms of the interface with the learning automaton situated therein. The automaton latches onto information from the universe (*e.g.* information is gained from sensing equipment, converted from an analogue to a digital signal, and stored in memory), in the form of a fixed length vector. The automaton later produces a fixed length output vector. The universe is affected in whatever way the governing laws of the universe determine (perhaps a solenoid changes state causing a chain of events). When the automaton checks for a new input vector, one will be there[1]. Goals should be implemented as an internal payoff function mapping input vector and internal state information into a payoff vector.

---

[1]Resources, such as those used by CS-1, are considered part of the universe, to be sensed via the input vector, and possibly changed via the output vector.

While the learning automata I have developed operate in a much more defined simulation, it remains a goal not to be too dependent on the particular nature of the simulation. There has been increasing interest in complex environments within the artificial intelligence community [Albers *et al.* 1985, Carbonell and Hood 1986, MacDonald 1986, Sammut and Hume 1986]. The simulations I have used in this research were designed to be simple and quick to run, tie in well with the work of other researchers, and be sufficiently general to effectively test the autonomous learning ability of the classifier systems I have constructed.

The environment I have chosen is discrete. It can be pictured as a network, with nodes representing valid positions, and edges representing accessibility. Nodes can be thought of as containing objects. Nodes have labels that may contain information about the node and its contents, or about other nodes in the world.

An automaton interacts with the environment in the following way: It can read the label of the node it occupies, and issue a (possibly vector valued) action code to the world simulator. In the actual programs implemented, the action value encoded a request to move along a particular edge (in Wilson's world, the request might not be honoured, in which case the automaton remains at its current position). Certain nodes are specified as goal nodes. When one of these is reached, the simulation is ended.

As can easily be seen, in my actual simulations, the universe is a much simpler, forgiving, and helpful place than it might be. The topology of the simulated world is basically a graph, and could in fact be totally modelled by an appropriate automaton (which would be a totally inappropriate way to think about the problem). The amount of time taken by the automaton to produce its action vector does not change the result of the action. Rewards need not be determined by the learning automaton.

Rewards are given from the environment based upon the state of the world. There is no requirement to breed learning automata that can produce their own goals from internal information. These rewards are only dependent on the state the automaton occupies, and not the particular path taken to arrive at that state. All the learning automata in this thesis would not work very well in an environment

where path dependencies were important for the reward gained, particularly those with internal world models.

Many of the environments used for research into classifier systems can be placed in this paradigm. Work by Booker [1982] and Wilson [1985] used world simulations based on an extension of a grid system (one hexagonal, the other square). Nodes in Wilson's world contained trees and food. Booker had nodes containing attractive and noxious substances, and nodes adjacent to these substances contained labels indicative of this. Nodes could be occupied either by a static item (*e.g.* an impassable barrier), an active item (*e.g.* the learning automaton itself), or could just be empty.

One of the important things for a system to take care of in a complex world is discarding irrelevant or overly noisy information. This noise can arise from various causes. There can be hidden variables to which the input stream to the program does not have direct access. Thus, multiple states of the world can give the same input to the learning automata. Another possibility is that the sensors of the system are inaccurate, and do not reflect the information available from the world. Another possibility is that the effectors of the automata are not consistent, producing different changes in the state of the world from those predicted. Finally, the world may not in fact be deterministic at all. It may just be that the world has noise as an integral part of its functioning. According to Kaelbling [1989], all such effects, excluding maliciously caused changes, can be handled by a stochastic model. As we would like to be able to handle ambiguous and noisy worlds, some form of stochastic model seems appropriate.

Some taxonomy of simulated worlds is desirable, so that we know what problems we have or have not solved. The ambiguity mentioned previously is one important attribute of a world. Another is repeatability. In other words can actions by the system lead to the state of the world being roughly the same. It should be pointed out that this has the potential to put the automaton into an infinite loop. Lack of repeatability means that once the automaton has made an error, it may not be able to backtrack.

Of the three finite state worlds chosen for empirical study: Node labels are unambiguous in N-World, rather ambiguous in Octworld and even more ambiguous

in Hexworld. Octworld and Hexworld are repeatable, whereas N-World is not repeatable. Each of these worlds is detailed below.

5.2 N-World

N-World is a graph having a variable number of edges per node. There are two possible starting nodes, and three goal nodes. It was purposely designed to show a significant flaw in current solutions to the credit assignment problem.

5.2.1 A Motivating Example

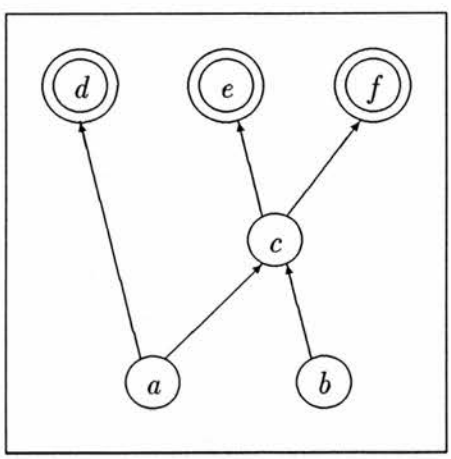

Figure 5–1: N-World.

Consider the simple finite state world described in figure 5–1. Circles indicate states of the world. Arrows indicate the ability to move from one state to another. Let us assign desirability to states d, e, and f, such that f is much more desirable than d, which is much more desirable than e (desirability is considered transitive). We start with a classifier system in which, for simplicity, there is exactly one classifier corresponding to each arrow (*e.g.* there is a unique classifier which only may become active in state a, that encodes an action which causes a move to state d).

A possible execution might start at a, and activate the classifier that results in a move to c (henceforth such a classifier will be termed $a\backslash c$). If the next classifier executed is $c\backslash e$, in a global reward scheme, because of the poor payoff associated

with e, both $c\backslash e$ and $a\backslash c$ are less likely to be activated in the future. Another execution, starting at b, might activate $b\backslash c$, and then $c\backslash f$. While $c\backslash e$ appropriately becomes even less likely to be activated due to the increased prominence of $c\backslash f$, nothing is done to improve the activation probability of $a\backslash c$, even though it has become clear that $a\backslash c$ was not only unjustly punished, but should indeed be rewarded (as it sets up $c\backslash f$). In the case of a subgoal reward scheme such as the "bucket brigade", one can demonstrate the same effect by assuming the existence of many states like e, reachable from c and with poor payoff. In an actual system over many runs the correct utility for each classifier might eventually be calculated, but speed of learning is an important issue, especially in light of the effects of incorrect classifier evaluation on the genetic component of the classifier system.

It should be mentioned that the classifier system itself cannot be certain that, for example, activation of $a\backslash c$ (when possible) puts the system in the same position as does the activation of $b\backslash c$ (again, when possible), because it does not have access to the state of the world. It has access only to its own sensors and internal state. Richer sensors imply a better correlation between world state and perceived world state, but bring with them the problem of determining which differences in perceived state are salient with regard to the problem domain.

5.3 Octworld, or Wilson's WOODS7

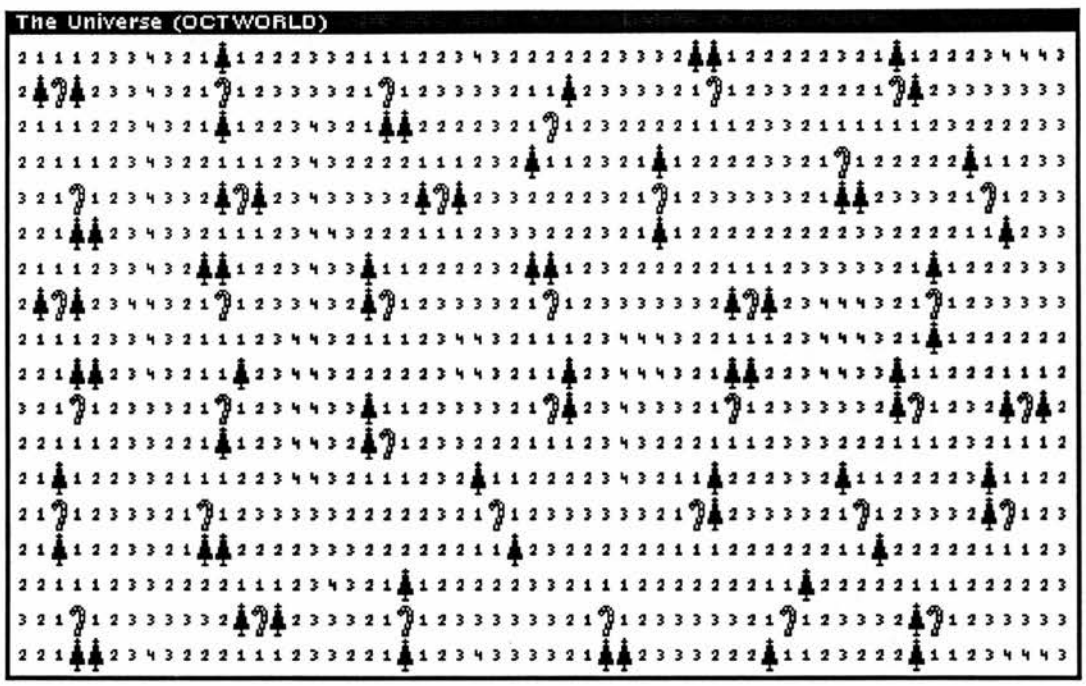

Figure 5–2: Octworld, with minimum distance to goal marked

Octworld (meant to be identical to Wilson's "WOODS7" [Wilson 1985]) is a simple two-dimensional world (figure 5–2). A rectangular (18×58) array is topographically extended in the natural way to a toroidal surface. Each element of the array is considered to be a node in a graph. Each node has eight edges linking it to the neighboring eight nodes. These eight possible exits can be thought of as the eight semi-cardinal directions (the four rectilinear directions plus the four diagonal directions). Note that moving in the diagonal directions is rather different from moving in the four standard grid directions. First of all, the distance travelled is $\sqrt{2}$ times the distance travelled in the rectilinear directions. A related difference is

that a diagonal move reveals information on five new surrounding nodes, whereas rectilinear moves reveal information on only three new surrounding nodes[2].

While most of Octworld is empty space, there are 37 clusters, each consisting of one goal node (Wilson's "food"), and two inaccessible nodes (Wilson's "trees") adjacent to the goal node. All empty nodes are potential start nodes.

Each node is labelled with information concerning the nodes immediately surrounding it. Each node can be described by a two bit quantity (Wilson describes these bits as "food smell" and "opacity"). An empty node is described by the bit pair, 00, an inaccessible node by 01, and a goal node by 11. The actual label given to a node is the string of bits consisting of the concatenation of the descriptions of all the nodes around it, starting from the node directly above, and proceeding counterclockwise. Each label is, therefore, a bit vector of length sixteen. The layout of Octworld is such that there are 92 distinct labels. Considering that there are 1044 nodes in the graph, knowing the label of a particular node still leaves a large amount of ambiguity about the precise identity of the node.

---

[2]Since a node in Octworld does not include information on its own contents, it could be considered another new node revealed by a move, but either way the number of nodes revealed is different depending upon the direction of travel.

5.4 Hexworld

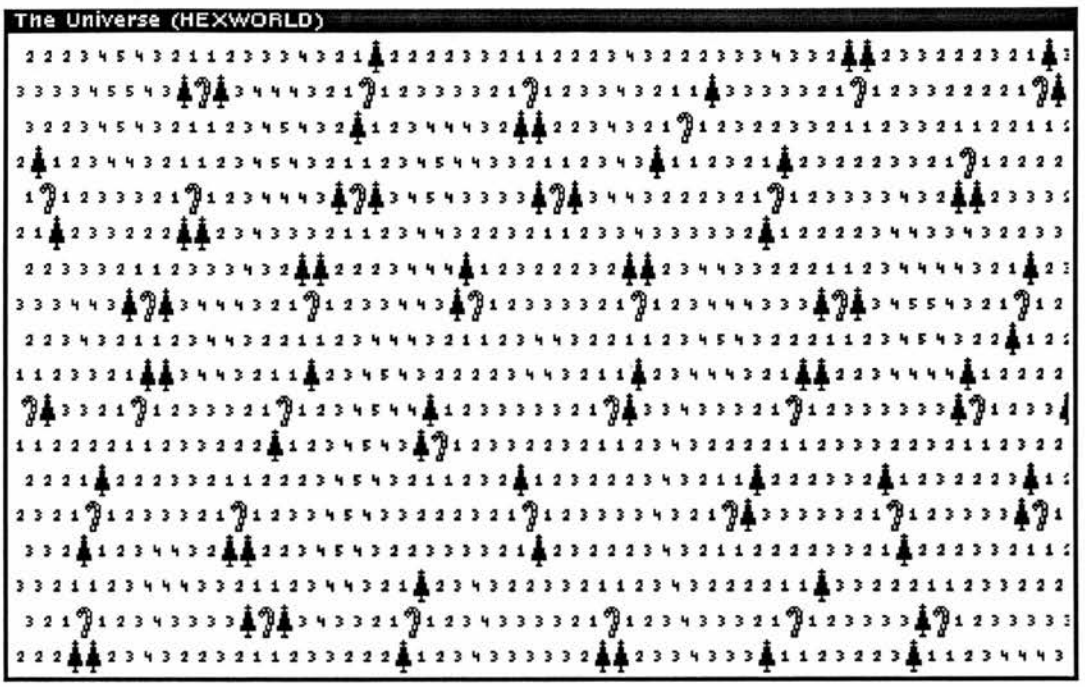

Figure 5–3: Hexworld, with minimum distance to goal marked

Hexworld is a simple variant on Octworld. The graph is changed such that two diagonal directions, corresponding to northeast and southwest, are no longer edges. The effect of this is to make the graph topographically equivalent to a hexagonal map (figure 5–3). This avoids the peculiarities of Octworld, where the diagonal directions exhibit markedly different behaviour than the four standard grid directions.

There remain 1044 nodes in the graph, but now there are only 53 distinct labels. Since there are fewer directions available for movement, the distance between any two points on the graph is at least as much as in Octworld. Because of this, one would expect a slightly worse average time to achieve a goal state in Hexworld. One property that is lost in Hexworld is that the pair of inaccessible nodes near goal states are no longer necessarily adjacent to the goal state.

Chapter 6

Classifier Systems that Look Ahead

6.1 Introduction

Looking ahead is the ability to predict, from a given situation, what new situation would occur if a particular action were performed. In order for looking ahead to be useful in deciding what action should be performed, the system must have some way of evaluating the situations predicted. Internal to a classifier system, a situation is the state of the message list. An action consists of posting a message. While there are some things about the future state of the message list that the classifier system can predict if a particular message is posted (*e.g.* the message list will contain the message just posted), it can not predict the environmental messages that may be placed on the message list by the sensors of the learning automaton due to the posting of that message. The traditional control strategy activates classifiers based entirely upon the ratings of the classifiers sensitive to the current message list. It does not base its decision upon the situation that will occur because of the activation choice that it makes. In order to keep the issues as clear a possible, I will concentrate specifically on predicting messages from the environmental sensors[1].

---

[1]I have not fully considered the issue of predicting non-environmental messages, or rather, of making use of the fact that the system can predict non-environmental mes-

Every time an automaton in any of the environments of the problem domain (see chapter 5) performs an action, it receives a new environmental signal, as well as reward information[2] (depending upon whether the automaton has occupied a goal node). In a standard classifier system, the automaton learns only from the reward information. It uses the environmental messages to shape its actions, but makes no explicit attempt to record causality. In particular, if there are no rewards available, the system learns nothing (there is a mild punishment normally administered, in the form of taxation, but if all movements are being punished, there is no real learning going on). When the automaton finally does get rewarded, only the classifier(s) that actually acted immediately preceding the attainment of the reward will get rewarded[3]. No other learning is accomplished. Likewise, on the next iteration of the system, no learning is accomplished until a classifier (or classifiers) either attains another goal, or sets up the classifiers that set up the last goal. After this point, the analysis becomes more complex, but it is still fairly certain that the system is not making use of all the information it has available.

One of the main difficulties with credit assignment in a classifier system is the delay between the activation of a classifier and the eventual reward from the environment. As stated previously, there is much more than reward information available to the system. Every time the system performs an action, the sensed environmental information changes. As this information accrues, an explicit and predictive world model can be constructed. After this has been done (or even as the model is being constructed), look ahead can be performed, incorporating the connectivity information obtained into the decision making process. This

---

sages. It does appear to be a useful extension to the lookahead techniques discussed in this chapter.

[2]In effect, the reward information is part of the environmental signal.

[3]This particular criticism is pertinent to subgoal reward schemes such as the bucket brigade. Global reward schemes instead suffer from the deficiencies that no learning occurs between rewards and that they may require an arbitrary amount of memory.

utilization of previously ignored information allows more rapid improvement in the performance of a classifier system.

The inability of a standard classifier system to learn by making use of environmental signals other than reward is caused by the internal world model of these systems. Standard classifier systems have implicit within them a purely reactive model of the world. The model is capable of suggesting an action, or set of actions, to be taken in particular situations. The rating of a classifier is indicative of how "good" it would be for the system to take the action specified in that classifier. What is absent from the model is any coherent justification for the action(s), or any predictions as to how the world will be changed by these actions. An internal world model allowing such prediction has many advantages. Learning can take place whenever the automaton receives an environmental message (particularly if it disagrees with the prediction), rather than only upon receiving the sparser external rewards. Long range planning can be performed. Actions can be justified to human supervisors, or explained to human students. The system can more easily accommodate changing goals since the world model remains valid even if the reward information becomes useless. An explanatory world model can also find solutions where a reactive model is either uninformed, or indeed misguided.

If the effect of the activation of a classifier is very accurately known, the automaton need not depend solely on the recorded merit of this particular classifier to determine whether the classifier should be activated. The control strategy for choosing classifiers for activation can include looking ahead to the new state of the world predicted by the classifier, and determining its desirability either by examination of the classifiers available from the predicted state, or by any recorded external reward attached to the predicted state. This evaluation criterion can be applied recursively.

Because the environmental message is so important to the lookahead classifier system, a simplified classifier system was easier to use as the basis for designing an explicit world model. I have used Wilson's Animat [Wilson 1985] as a basis for these simplifications. The main variations from a standard classifier system are the reduction of the message list to a single environmental message posted after

each external action of the system, and having classifiers compete to produce a single external action, rather than posting messages.

Before introducing the techniques used in my research, some attention needs to be paid to existing classifier systems that are not entirely reactive. Gofer [Booker 1988, Booker 1982] and Gofer-1 [Booker 1989] show some predictive ability, but the prediction is of rewards rather than world states. This is a very small subset of the changes actually caused by activation of a classifier. Correspondingly only some of the advantages described above would be obtained (chiefly learning via unreliability of prediction). CFSC2 [Riolo 1990], on the other hand, is capable of predicting world states and using lookahead to influence classifier selection. Due to the recency of the work, I was unable to either attempt a rational reconstruction for comparison with my own research or incorporate the many good ideas presented. A brief description of CFSC2 with its strengths and weaknesses compared to my own systems is included in section 8.3.

6.2 An Explicit World Model

While deterministic finite state worlds occur commonly in machine learning literature ([Booker 1982, Grefenstette 1988, Holland and Reitman 1978, Wilson 1985], to name just a few), ideally, one would like to choose a modelling scheme that is capable of modelling any possible universe[4]. In practice, however, any modelling scheme is limited by the representation. Certainly our learning automaton should be able to handle deterministic finite state worlds. We cannot just assume, however, that even a deterministic world will appear deterministic to the classifier system. Full state information is not, in general, available to the learning automaton, so distinguished states in the world may not be easily distinguishable to the learning automaton. For this reason a representation capable of handling a non-deterministic model is essential. Kaelbling [1989] shows that non-determinism

[4]Westerdale [1990] discuses "Queasy-Morphisms" and Rosenschien talks about mapping world states to machine states [Rosenschein and Kaelbling 1987, Rosenschein 1985].

is capable of handling various complications. So, a non-deterministic finite state automaton (NFA) appears a good model of the world to start with.

The most obvious way to model an NFA is to construct a state space and transition function [Hopcroft and Ullman 1979]. In a classifier system, the most obvious internal representation of world state is the message list. Given the simplistic classifier system under discussion, the message list simply consists of the current environmental message. This message is binary so this leaves our NFA with $2^{(message\ length)}$ distinguishable model states (or situations). The actual world occupied by the learning automaton, of course, might have any number of states (*e.g.* uncountably infinite). Some model states may never be reached (since the environment may never produce the corresponding message), and many of the model state distinctions made may be irrelevant to the optimal functioning of the learning automaton.

For an NFA, the transition function takes a state and an input[5], and produces a set of states that can follow. We really would prefer a more detailed model than this, since one would expect some states more likely to follow than others. We therefore want an extended transition function which produces a probability distribution over the state space, giving the likelihood of any particular state occurring following execution of the specified action in the specified state. In short, we want to create a Markov model of the world. For this we need to form a probabilistic mapping between states, actions, and other states. Classifiers in the system already associate situations with actions, and situations are model states. If we associate with each classifier a probability distribution of situations that occur after activation of that classifier (*i.e.* the distribution of situations that follow when the action specified by that classifier is executed), classifiers can be used to define transition functions. Using classifiers allows generalization as well, since a classifier tends to be appropriate for more than one situation. It is

---

[5]In the case of our explicit world model, a state in the NFA corresponds to a situation (or model state), and an input for the NFA corresponds to an action that may be taken by the automaton.

proposed here to obtain the transition information from empirical operation of the system. When a classifier is activated, and its action performed, the situation that follows is also recorded and associated with that classifier. In the history of many executions of the same classifier, different situations may be encountered. Some summary of this information must be maintained with each classifier. The transition function is therefore likely to become nondeterministic. Another cause of nondeterminism is that multiple classifiers specifying the same action may match a given situation. In such cases all of these classifiers should be taken into account in the definition of the transition function. Due to the empirical nature of the transition information, some attempt should be made to take into account the uncertainty of this information. One of the important distinctions between the system described in section 6.3 and those described in section 6.4 is their approach to this problem.

6.3 Lookahead-Average

Lookahead-Average (La-Av) is a classifier system I devised based upon my reconstruction of Wilson's Animat. The major difference between La-Av and the reconstructed Animat is simply the use of lookahead information (with all of the accounting this implies) as an aid to classifier evaluation. There are other differences as well, and these too will be described. The general algorithm is as follows:

1) Create initial set of classifiers. The current action set is empty.

2) The autonomous learner is placed in the virtual world with an empty message list.

3) The input processor adds a message to the message list based upon its sensory equipment.

4) Determine classifiers eligible for activation given the current message list.

5) Determine relative priorities of eligible classifiers based upon lookahead information. Determine probabilities for taking each of the actions available.

6) Choose probabilistically amongst the actions available. A new current action set, consisting of all eligible classifiers that specify the action chosen, is formed. The previous "current action set" is still of interest, and is referred to as the previous action set.

7) Accounting is done for all of the classifiers in the system.

   7.1) *Strength* is taken from members of the previous action set, and is distributed evenly between members of the current action set.

   7.2) A distance estimate is taken via examination of the current action set, and this is used to update the *distance* values of the previous action set.

   7.3) All classifiers are taxed a percentage of their *strength* values.

   7.4) Classifiers from the previous action set have their world model information updated to include the message produced by the input processor.

8) The output processor updates the effectors of the automaton based upon the action selected.

9) Reward information is made available to the automaton (*i.e.* whether a goal node has been reached and, if so, the magnitude of the reward).

10) If a goal node has been reached, then:

   10.1) The reward is distributed evenly between members of the current action set.

   10.2) The members of the current action set have been shown to be a single action away from a goal. The *distance* values of these classifiers are updated in order to reflect this fact.

   10.3) Classifiers from the current action set have their world model information updated to include the reward gained.

10.4) Possibly perform a genetic operation (recombination, duplication, intersection and mutation[6] are supported).

10.5) Go to step 2

Otherwise, go to step 3.

The key differences between La-Av and the reconstructed Animat are the modifications as to how actions are selected (steps 5 and 6), and the insertion of world model updates (steps 7.4 and 10.3). Step 5 now takes into account information based upon lookahead (and computes the probabilities of action selection differently from a linear combination of classifiers supporting each action, which in turn causes the minor modification in step 6), and steps 7.4 and 10.3 accumulate the information required to make this possible. What is required is sufficient information to predict what will happen (from the viewpoint of the learning automaton) if a particular course of action is taken by the learning automaton and a method of rating such potential happenings. The methods chosen for this in my research involve attaching extra information to the classifiers within the classifier system. In La-Av, the form of this information is a bag[7] of messages and reward information. Given the architecture of the reconstructed Animat, the message from the environment is the only message in the message list, and is equivalent to a situation or model state as discussed previously in this chapter. When a classifier has its action chosen by the control algorithm, the environmental message that follows is stored in the bag associated with the classifier. Since many classifiers may be in the action set, the same message may be put in many bags. If one wishes to determine what situations have followed activation of a particular classifier (the entire current action set is considered to be active), it suffices to examine the bag associated with that classifier. Since it is a bag, and not a set, the relative

---

[6]This mutation is different from that of the reconstructed Animat (see section 4.5), but equivalent to the one in Q-CS.

[7]I am using bag in the technical sense of being similar to a set, but allowing multiple occurrences of elements.

frequency of situations previously encountered can be used as an estimate of the actual probability distribution for situations following activation of the classifier in question. In La-Av, if a reward is encountered by the learning automaton, this reward, rather than the environmental message, is placed in the bag. This is due to the fact that in all of the virtual worlds encountered, the simulation ends when a reward is received. Thus, the only salient information about the state is the reward received, since no decisions need be made in a rewarding state, and no lookahead can be done from such a state[8]. In the general case, the reward information would have to be considered part of the environmental message, and stored as a single entry in the bag.

From the viewpoint of implementation, this scheme means maintaining a set of pairs where one item in the pair is the situation, and the other, the number of occurrences of this situation in the bag. While this might require $2^{(message\ length)}$ pairs for each classifier in the system, in the worlds investigated the number of pairs required was much smaller[9]. Rewards were treated as special situations that were distinguishable from non-reward situations. Reward situations were distinguishable from each other only by the difference in magnitude of the reward. As in the case with the number of messages, the virtual worlds encountered contained few different reward values, the highest number of types being encountered in N-World, where there are three different reward values.

---

[8]If there were means-ends analysis being performed, the information could be useful, but La-Av does not perform such analysis, nor does such analysis seem useful in the virtual worlds being investigated.

[9]For example, in Octworld there are only 92 situations potentially distinguishable by the classifier system, out of a possible 65536. It is interesting that the actual number of nodes in Octworld, 1044, is not important in the analysis, although that number might become more significant if internal messages were available to the automaton, since one might hope that the number of model states encountered would tend towards the number of states in the world.

The above is sufficient to describe what is required in steps 7.4 and 10.3. Step 5 computes the relative priorities of classifiers based upon this information. The lookahead rating of a classifier is the average rating of all the situations and rewards encountered immediately after this classifier was in the action set[10]. In other words, the situations and rewards stored in the bag associated with the classifier are evaluated and the average obtained thereby (plus an uncertainty factor) is the lookahead rating of the classifier. The rating of a reward is simply the magnitude of the reward. The rating of a situation is the rating of the highest rated classifier (in the Animat sense of rating, *strength/distance*) with a taxon matching that situation.

Initially, each classifier can make no prediction about the situation that may arise due to its activation. In order to handle this case, an estimate must be made of the value of activating an untried classifier. Further, the prediction made by a classifier that has been activated twice should be considered more reliable than another that has been activated but once. The way I accomplished this in La-Av was that whenever the lookahead rating of a classifier was being calculated, an extra value was included in the average: an expected value for the unknown. The lookahead rating of an untried classifier would be exactly equal to the expected value for the unknown (since the bag associated with this classifier is empty). For a classifier that had been tried only once, the expected value of the unknown would contribute half of the lookahead rating of the classifier (since there is one element in the bag plus one element representing the unknown). Likewise, a classifier that has been tried n times will have n elements in its associated bag, and the estimate for the unknown will be weighted by $1/(n+1)$ in computing the lookahead rating of the classifier (there is nothing special going on here, this is just the standard way of computing averages). This decrease in importance of the expected value of the unknown as the experience of a classifier increases is exactly what we wanted. The magnitude chosen for the expected value of the unknown will strongly affect

---

[10]Another number is also included in this average to represent uncertainty. It will be discussed shortly.

the investigative tendencies of the resultant automaton. In the case of La-Av, the expected value of the unknown is defined to be equal to the average rating (*i.e. strength/distance*) of all classifiers in the system. This appears a reasonable first estimate of the performance of a random action in a situation chosen at random. At one point I experimented with a classifier system designed to be more investigative, called La-Max. In La-Max, the expected value of the unknown was set equal to the rating of highest rated classifier in the system. The empirical results for La-Max in Octworld were not encouraging, and a theoretical analysis of its performance, compared to La-Av, in N-World was unpromising, so La-Max was dropped from study[11].

There are several other distinctions between La-Av and the reconstructed Animat. In hindsight, I expect these have crossed the wrong side of the line between innovation and obfuscation, since they interfere with direct comparison of lookahead versus non-lookahead systems. Non-random creates have been removed (random creates are used in any situation that would have invoked a non-random create). I think this is just as well, since non-random creates would not be available in real world situations. The initial classifiers in La-Av do not contain any *distance* values[12]. I think this is a minor improvement, since these *distance* values must otherwise be set with no empirical basis, but is unlikely to have a major effect. The other change, however, was in action selection (steps 5 and 6), and the effect of this change is difficult to estimate.

For the purpose of action selection, instead of treating classifiers that match the current situation and specify the same action independently, they are treated

---

[11]In fact, from the empirical evidence, it seems that the value of the unknown chosen for La-Av is still higher than is desirable for the virtual worlds encountered. Part of the reason for this is that classifiers with low *strength* are constantly being pruned from the classifier system.

[12]Until the classifiers gain a *distance* value, they do not contribute to distance estimates made by the automaton. The rating of such a classifier is just its *strength* value (this is equivalent to a *distance* value of unity, which is the value used by Wilson).

as groups. From each group, the classifier with the highest rating (on the basis of lookahead information) is is used to determine the relative probability of this particular action being taken. This scheme was designed to keep a large set of classifiers that were on the verge of zero *strength* from overpowering by shear numbers, a classifier more highly rated. I can not say that I am convinced whether or not this is a more effective technique than the one used by Wilson.

All in all, La-Av is a lookahead variant of Wilson's Animat. The main innovation is the world model and the handling of uncertainty. The use of bags to record the results of past activations provides sufficient information for a good probabilistic transition function, and uncertainty is handled in a plausible and flexible manner.

6.4 Q-Classifier and Dyna-Q-Classifier Systems

The Q-Classifier and Dyna-Q-Classifier Systems (Q-CS and Dyna-Q-CS) are more innovative systems than those previously discussed. They depart in two important ways from all classifier systems of which I am aware. First they use an adaptation of Q-learning [Sutton 1990, Watkins 1989] instead of a bucket brigade. Also, the actions are chosen in a novel way. As in La-Av, classifiers sensitive to the current situation, with the same action, are not treated independently. A form of intersection of the events recorded by all of these classifiers is taken, allowing added accuracy for the predictions of events to follow.

Q-learning was developed by Watkins, and has been extended by Sutton into a powerful incremental planning scheme (Dyna-Q). It was the incremental planner that motivated me to adapt Q-learning to the classifier system domain. Dyna architectures attempt to increase the learning speed of machine learning algorithms by interleaving execution of the algorithm within the environment and execution within an internal model of that environment. Thus, whenever processor time can be spared, a situated autonomous learner can consider situations that may not be related to the current situation, and, in effect, plan ahead for such situations. In a dangerous environment, such planning ahead may avert catastrophes

(*e.g.* falling off a cliff is normally less dangerous within a simulation). Sutton has applied the Dyna architecture to Q-learning to produce Dyna-Q. Q-learning is particularly amenable to the Dyna architecture, because Q-learning converges no matter which order environmental states are investigated or what actions are chosen by the learning automaton (as long as all possible decisions at all states are investigated a sufficient number of times). This in turn means that model states can be investigated in any order, without impairing the performance of Q-learning. This is not true in the case of the bucket brigade.

The bucket brigade is not well suited to an incremental planning scheme like Dyna. The problem is that the bucket brigade operates as an economy where value is conserved. If the system determines that classifier a is strong, and classifier b sets up classifier a (this being determined by the explicit world model), it seems reasonable to strengthen b, but in a bucket brigade, this requires taking strength away from a. In the normal course of events, a would fire next, and presumably gain strength, but if one does not wish to be forced to follow the control strategy all the way to the goal, the bucket brigade ends up punishing the classifiers active at the point where the standard control strategy is abandoned. Even in a bucket brigade system that has reached an equilibrium, not following a decision policy to its conclusion can be detrimental to future decisions.

Before examining Q-CS and Dyna-Q-CS, it is useful to examine pure Q-learning in more detail. In situated autonomous learning systems there is a policy for choosing what action to take, and an update equation for modifying this policy. In Q-learning, this is done by keeping an estimate for every model state, x, and action, i, of the value to the automaton of taking action i in state x. This estimate is known as Q_{xi} (hence Q-learning) and is used to influence the decision policy of the automaton. The value of a state, x, is known as $e(x)$ which is defined as follows:

$$e(x) = \max_{i \in A_x} Q_{xi}, \tag{6.1}$$

where A_x is the set of actions available from state x. In other words, the value of a state is the value of the best action available from that state. Integral to Q-learning is the concept of discounted future reward. In a deterministic world, this

would just be the reward for taking a given action in the current state plus some constant fraction of the value of the next state. In a nondeterministic world, we must deal with expected values. It is desired that each Q_{xi} approach the expected value of the discounted future reward for taking action i in state x. That is,

$$Q_{xi} \text{ should approach } E\{r + \gamma e(y) \mid x, i\}. \tag{6.2}$$

In other words, we would like Q_{xi} to be the expected value of the reward, r, for taking action i in state x plus some constant discount ($0 \leq \gamma < 1$) of the value of the next state, y[13]. Remember that we are dealing with probabilistic transitions here, and neither r nor y are necessarily determined by x and i. Neither are r and y themselves necessarily linked in any way, so the solution may be problematic. In order to achieve this goal, an update function exists,

$$Q_{xi} \leftarrow Q_{xi} + \beta(r + \gamma e(y) - Q_{xi}), \tag{6.3}$$

where β is a learning rate parameter (normally $0 < \beta \leq 1$). So, every time action i is taken in state x, the value Q_{xi} is changed to reflect the reward received (this comes from the environment) plus the discounted value of the new state occupied by the automaton (this is computed via equation 6.1). How drastically this new estimate affects Q_{xi} is determined by the parameter β[14]. With the given range for β, the Q values are guaranteed to converge to the desired values, so long as all actions from all states are tried a sufficient number of times.

Now that we have defined these estimates, Q_{xi}, the automaton needs to have a policy for action as well. The most obvious policy for any state, x, is to choose the action with the largest Q_{xi}. If the Q values are consistent with the world being modelled, the most obvious policy is also the best policy. However, before the Q

---

[13]The discount factor, γ, insures that the Q_{xi} and $e(x)$ values are finite, even in worlds where the learning automaton is free to move back and forth between two rewarding states.

[14]If β is zero, no learning takes place. If β is one, the old value of Q_{xi} is entirely ignored in the update process.

values reflect the world being modelled, this policy would be too rigid to allow adequate exploration of the state space, so Watkins used a probabilistic selection mechanism,

$$P(i|x) = \left[e^{\alpha Q_{xi}} \Big/ \sum_{j \in A_x} e^{\alpha Q_{xj}} \right]. \tag{6.4}$$

The probability of choosing action i, given that we are in state x, is exponential on Q_{xi}. The constant, α, is an annealing quantity that approaches infinity over time. As this happens, the differences in Q values become ever more important. The appropriate rate of increase for α, is, however, problematic.

One disadvantage of Q-learning over classifier systems is that Q-learning requires a Q value for every state-action pair. In a classifier system, there is a strength associated with every classifier. Since classifiers can be sensitive to many different states, the storage capacity of the learning automaton is not strictly dependent on the number of states in the world. A classifier system will make do with as much (or as little) storage as one cares to allocate it. Thus, there is good motivation for incorporating the advantages of Q-learning within a classifier system.

Q-CS

Q-CS is a system that exhibits the advantages of Q-learning within a classifier system. The basic algorithm for Q-CS is very similar to that of Wilson's Animat or La-Av.

1) Create initial set of classifiers. The current action set is empty.

2) The autonomous learner is placed in the virtual world with an empty message list.

3) The input processor adds a message to the message list based upon its sensory equipment.

4) Determine classifiers eligible for activation given the current message list.

5) Combine eligible classifiers into groups supporting different actions, determining the probability for taking each action available (via equation 6.13).

6) Choose probabilistically amongst the actions available. A new current action set, consisting of all eligible classifiers that specify the action chosen, is formed. The previous "current action set" is still of interest, and is referred to as the previous action set.

7) Accounting is done for all classifiers in the previous action set. The world model information for each is updated to include the message produced by the input processor, and then the strength values are recomputed (via equation 6.6).

8) The output processor updates the effectors of the automaton based upon the action chosen.

9) Reward information is made available to the automaton (*i.e.* whether a goal node has been reached and, if so, the magnitude of the reward).

10) If a goal node has been reached, then:

    10.1) Update the world model information of classifiers in the current action set to include the reward gained, and then recompute their strength values.

    10.2) Possibly perform a genetic operation (recombination, duplication, intersection and mutation[15] are supported).

    10.3) Go to step 2

    Otherwise go to step 3.

The main differences between Q-CS and La-Av, Animat, occur in steps 5, 7 and 10.1. Aside from using a credit assignment algorithm based upon Q-learning, the mechanism for recording and utilizing the world model is different. This mechanism is also foreign to Q-learning, and therefore complicates what would otherwise be a more parallel description.

---

[15]This mutation is different from that of the reconstructed Animat (see section 4.5), but equivalent to the one in La-Av.

First, a few definitions:

M_x is the set of classifiers with taxons matching state x.

M_{xi} is the set of classifiers in M_x specifying action i.

S_c is the strength of classifier c.

F_c is the probabilistic followset of classifier c.[16]

A_x is the set of actions available in state x.

The variables a, b, and c are used for classifiers, f is used for followsets, x, y and z are used for model states (situations), i, j and k are used for actions, and p is used for events.

In Q-CS, classifier strengths take the place of Q values. Q-learning requires an evaluation function for states and an update function for Q values. In Q-CS this translates to requiring an evaluation function for situations and an update function for classifier strengths.

In Q-learning there is a single scalar value, Q_{xi}, for every state and action in the world model. In a classifier system, there is a set of classifiers, M_{xi}, associated with each situation and action. Each of these classifier will have its own strength value. Some function must be applied in order to take this set and produce a single value that can be used in place of Q values. The function chosen for Q-CS was the average strength[17] of the classifiers in M_{xi}. Thus the parallel to equation

---

[16]This will be described in more detail later in the text.

[17]As with La-Av, this method of combining M_{xi} (in the terminology used for La-Av, those classifiers that match a situation and specify the same action) discriminates against large groups with low valued classifiers. While, as in the case of La-Av, I am not convinced that this is desirable, there is more justification for doing this in Q-CS. A simple additive scheme would destroy the discounted future reward notion inherent in Q-learning. Using the arithmetic mean appropriately scales the evaluation function.

6.1 for Q-CS is:

$$e(x) = \max_{i \in A_x} \left[\left(\sum_{c \in M_{xi}} S_c \right) \Big/ |M_{xi}| \right], \tag{6.5}$$

where $|M_{xi}|$ is the number of elements in M_{xi}.

The update function for classifier strengths in Q-CS is handled in a way reminiscent of Q-learning, but modified to incorporate the portion of the explicit world model contained within a classifier. Rather than updating the strength of a classifier directly, Q-CS uses the (as yet undefined) followset of the classifier as the basis for updating classifier strength. The strength of a classifier is not incrementally changed with each update. It is evaluated anew from the information in the followset of the classifier[18]. The strength of a classifier in Q-CS is set to an average of the rewards and discounted estimated utilities of situations in the followset of the classifier. This is parallel to Q-learning. In order to define the update function for classifier strength, we need to better define a followset.

The idea behind the followset, as in the bag used by La-Av, is to keep a record of the situations and rewards encountered after activation of that classifier. The followset, however, contains timing information as well. That is, it not only records what events occurred, but also when they occurred. This allows information from multiple followsets to be combined more accurately. Events duplicated between multiple followsets can be identified, and events noted by one followset can be shown definitely not to be contained within some other followset. In order to accomplish this, a followset must also have information as to when it started recording events. From a more technical standpoint, then, a followset consists of an origination time stamp, and a set of events. An event is a pair consisting of either a time stamp and a situation or a time stamp and a reward value. A followset is associated with each classifier. When the automaton chooses an action, every classifier in the action set is updated as follows: If the action produces a

---

[18]In a sense, this is a fundamental departure from Q-learning, since the update function is not incremental. Contrariwise, the same actual value desired in Q-learning is just being calculated a different way. It also allows changes in evaluations of classifiers and states to propagate more quickly.

reward from the environment, the classifier has the value of the time counter and the reward added as a pair to the followset of the classifier, otherwise the time counter and resulting situation are added. The time counter is then incremented. The origination time stamp for a followset is the value of the time counter when the followset was created[19]. In the case of the followset of a classifier, it is the value of the time counter when the classifier was created[20].

The equation to update the strength of a classifier will be constructed a piece at a time, using auxiliary functions that will prove useful later on. The reason for this approach is the lookahead nature of the update function, which complicates the parallels to Q-learning. The strength of a classifier is updated to be some function of its followset.

$$S_c = evalset(F_c), \tag{6.6}$$

where *evalset* is a function that determines the value of a followset. The function *evalset* is defined as:

$$evalset(f) = \left[\sum_{p \in f} evalevent(p) \Big/ |f| \right], \tag{6.7}$$

where $|f|$ is the number of events in f (*i.e.* the value of a followset is just the average value of the events contained). The value of an event depends upon whether it is a reward event or not. If the event was a reward event, then the value of the event is just the value of the reward. Otherwise, the value of the event is the discounted value of the situation associated with the event.

$$evalevent(p) = \begin{cases} reward(p) & \text{if } p \text{ is a reward event} \\ \gamma \times eval(situation(p)) & \text{otherwise} \end{cases}, \tag{6.8}$$

where *reward* is an access function returning the reward of an event, and *situation* is an access function returning the situation of an event. We already have a situation evaluation function: e (see equation 6.5). We could use it as *eval* in

---

[19]Thus, followsets that have no events recorded within them can still be distinguished.

[20]At the beginning of each run, the time counter is set to zero.

equation 6.8, to produce a simple *evalevent* function:

$$simpleevalevent(p) = \begin{cases} reward(p) & \text{if } p \text{ is a reward event} \\ \gamma \times e(situation(p)) & \text{otherwise} \end{cases} . \quad (6.9)$$

While such a function is used in Q-CS, it is used as only part of the evaluation function. In order to provide a more accurate evaluation of the utility of a particular action in a given state, I devised a more complex scheme.

To date most work in classifier systems has tended to treat classifiers individually. One of the interesting features of Wilson's Animat was its conglomeration of all classifiers that matched the current situation and had the same action into a single action set. While this didn't change the action immediately taken by the automaton, it allowed the entire action set to be updated by the results of the action. My work extends this concept further to actually change the rating of an action in a non-cumulative way.

Every time an action is chosen by a system, the resulting reward or situation is recorded as an event associated with each classifier of the action set. The simple evaluation scheme described by equation 6.1 makes an implicit assumption of independence between the classifiers of the action set, but independence is highly unlikely. Two classifiers that are part of the same action set were probably in the same action set previously. The events recorded in the followsets of these classifiers are likely to overlap. This overlap can be very complex, but there is information there to be extracted.

At first it seemed best to take M_{xi} as a pool of knowledge as to what would happen if action i was taken in situation x. In this light it appeared that events redundantly represented in the classifiers of M_{xi} should be viewed as merely one occurrence. That this is incorrect is easy to show. If one finds an action set where one classifier is a specialization of another classifier, the events in the followset of the specialist will always be contained in the followset of the generalist. By counting events only once, the specialist is ignored, yet it is the specialist that has the more salient experience for the current situation. So, rather than a union of all events within the followsets of the action set, an intersection between the followsets is more appropriate. A simple summation of events shows a tendency in the appropriate direction, but this seems like a crude approximation of the

intersection operation available. The idea is to synthesize the followset of an even more specific classifier than any that may actually be present in the system, one that incorporates as much of the information contained within the system as can be brought to bear. We need to define an operator capable of doing this. To put it in a mathematical form, we are looking for an operator, I_{xi}, the intersection of the followsets of all classifiers in M_{xi}:

$$I_{xi} = \bigcap_{c \in M_{xi}} F_c. \tag{6.10}$$

This intersection operator in the above equation is not a standard one. This operator must make use of the various time stamps in the followsets.

Within a classifier system, new classifiers are being continuously created as old ones are destroyed. New classifiers have empty followsets. If we used standard intersection as the operator on followsets, I_{xi} would be empty when the new classifier is in M_{xi}. All information gained about M_{xi} previous to the introduction of the new classifier would be lost. This is far from desirable. In order to avoid this loss, all classifiers have a time stamp dating their creation. Intersection of two followsets is defined as follows: Without loss of generality, assume that the first followset has a creation stamp predating that of the second followset. The intersection of the two followsets consists of the creation stamp of the first followset, and the union of all elements of the first followset that predate the creation stamp of the second followset and the set intersection set of pairs of each followset.

I can now define the more complex *eval* function used in Q-CS,

$$eval(x) = \max_{j \in A_x} \left[\left(\sum_{p \in I_{xj}} simpleevalevent(p) \right) \Big/ |I_{xj}| \right]. \tag{6.11}$$

The function *simpleevalevent* was used instead of *evalevent* to avoid an infinite regression.

This finally puts us in a position to specify what exactly happens in step 7. For each of the classifiers in the previous action set, update the followset by including the current event, and update the strength value via equation 6.6. Likewise for step 10.1, add the event to classifiers in the current action set, and re-evaluate their strengths.

The parallel equation to the action selection policy of Q-learning (equation 6.4) would be:

$$[P(i|x) = \left[e^{\alpha evalset(I_{xi})} \Big/ \sum_{j \in A_x} e^{\alpha evalset(I_{xj})} \right]. \tag{6.12}$$

Because of the lack of fast floating point, the exponential probabilities assumed by Watkins were not used for action selection in my implementation. Instead, the function used for Q-CS was:

$$P(i|x) = (evalset(I_{xi}))^3 / \sum_{j \in A_x} (evalset(I_{xj}))^3. \tag{6.13}$$

The cubic was chosen as a rapidly increasing function that could be computed speedily (with no floating point arithmetic required). Equation 6.13 defines the selection process of step 5 in the Q-CS algorithm.

In step 1 of the algorithm, 400 classifiers are initialized. Taxons and actions are initialized as per the reconstructed Animat and La-Av. Each is created with a time stamp of zero, and an empty followset. The strength value is set to a special value which means that the classifier has not yet been tried. For the purposes of equation 6.5 such a classifier is not considered part of M_{xi}.

There are a number of constants and special cases in Q-CS. The constant γ was set equal to 0.5. The size of followsets was limited to ten events (as each run has large numbers of events, some arbitrary limit on followset size had to be made). When no estimate existed for the utility of an action in a particular situation, some default utility had to be found.

When an event was to be placed on a followset that already contained ten events, the oldest event in the followset is lost. The origination time stamp of the followset is changed to match the time stamp of the event being deleted. The new event is then added to the followset.

The probability function given in equation 6.13 does not define what happens when I_{xi} is empty. First of all, for i not in A_x, $P(i|x) = 0$. If i is in A_x, then one of the two following hold. If $\forall i \in A_x$, I_{xi} is empty, then $P(i|x) = (|M_{xi}| + 1)/(|M_x| + |A_x|)$. Otherwise, the average of the cubes of all the defined $evalset(I_{xi})$ is calculated, and $P(i|x)$ is defined as if all actions with empty I_{xi} had $evalset(I_{xi})$ set equal to the cube root of this average.

A number of changes were made with respect to the genetic algorithm in Q-CS compared to that of La-Av or the reconstructed Animat. In particular, all new classifiers, except for those produced via duplication, start off with a time stamp reflecting the current time (just the number of actions taken by the automaton since it was created) and a strength set to the a special value (as described above for the initialization of the classifier system).

In order to stimulate the diversity of classifiers, whenever an action i was in A_x, but M_{xi} was empty, there was a 0.02 chance that a new classifier would be generated, equivalent to Wilson's random creates. The concept of the combined set of classifiers having too low strength, which was used by Wilson to trigger this operation, makes little sense in the context of Q values.

The other change in the genetic algorithm concerned deletion. Classifiers were chosen for deletion with probability proportionate to their age. The age of a classifier being the difference of value of the time counter (which is just a count of the number of actions taken by the automaton since its inception) and the time stamp of when the classifier was created[21]. Unity was added to this difference so that all classifiers had a finite chance of being chosen for deletion. The reasoning behind this choice is as follows: Classifiers maintain world model information, but if the world changes, this old information might be detrimental. If a classifier is useful, then it will be reproduced in the younger population, so, the loss of useful genetic material should remain small.

Dyna-Q-CS

The main point of Dyna architectures is to incrementally update the world model of a learning algorithm without resorting to real world trials. In a standard application of the Dyna architecture, hypothetical situations from the world model would be presented to the learning algorithm along with actual situations. In Q-learning, a hypothetical state from the world model would be chosen, and an

---

[21] Note that this is a different time stamp from the one in the followset of the classifier, because the time stamp in the followset may change.

action selected. The normal decision policy of a learning is probably too conservative for hypothetical situations. Sutton [1990] demonstrates that choosing an action uniformly from those available is an effective policy for responses to hypothetical situations in the implementation of Dyna-Q. The Q value for that state and action would then be updated via equation 6.3, where r and $e(y)$ are determined by the world model.

If I had precisely paralleled Dyna-Q in my implementation of Dyna-Q-CS, a hypothetical situation would be chosen, the set M_x computed, an action, i, selected, and those classifiers in M_{xi} would be updated via equation 6.6. In Q-CS, this is problematic to do, and there is a better way to proceed.

The problems are of two kinds. First, the world model in Q-CS loses states. The followsets are limited in memory (in the simulation used, only the most recent 10 events were maintained for each followset). This means that there may be useful classifiers that are not sensitive to any situation currently known to the world model. Such classifiers would never be chosen to be updated. Second, computing M_x is expensive, to say nothing of computing a decision policy. If we were forced to do things in such a fashion, we could, but it is worthwhile to investigate alternatives.

In Dyna-Q, the whole purpose of the entire exercise of choosing a hypothetical state and an action is to determine which Q value to update. One could just select a Q value to update at random. In effect, choosing a state, x, and selecting an action uniformly at random from the set available is just that.

In Q-CS, classifiers take the place of Q values. All that is needed in order to make Q-CS into Dyna-Q-CS is a scheme for updating classifiers which are not members of M_{xi} for the current state, x and chosen action, i (*i.e.* classifiers that are not members of the current action set). Dyna-Q-CS performs its Dyna operations in parallel with the evaluation of those classifiers in the action set (as part of steps 7 and 10.1). A fixed number (in the case of the simulation data, five) of classifiers are chosen at random, and their strengths readjusted via equation 6.6. Because of this, information from arbitrarily far away in the model space can be incorporated throughout the model, just as it is in Dyna-Q.

The technique of using a random choice of hypothetical states has been criti-

cized when viewed from the standpoint of a standard Dyna-Q system, because the space of nodes to be updated is one-to-one with the nodes in the world. Since this is potentially enormous, it is often thought that some more intelligent technique for choosing values to update is required. In Dyna-Q-CS, what need updating are classifiers, and the designer of the classifier system has chosen the number of classifiers the system will have. Nevertheless, if this is still considered a problem, Dyna-Q and Dyna-Q-CS do not theoretically require that the things to be evaluated be chosen at random. Any scheme (*e.g.* forward-chaining from the current state, backward chaining from known goals, or both) can be used for this selection.

The use of forward chaining and dynamic planning, rather than more conventional forms of planning, was dictated by my original design goals. An automaton operating in an unstructured and asynchronous environment such as the real world must be able to react instantaneously. With a conventional planner much information would be lost as the situation changed, and things would have to be replanned. While my particular implementation of forward chaining is rigid in this regard, forward chaining allows an incremental improvement in assessing the correct response, and therefore an immediate reaction is always available. Likewise, incremental planning can be given as much processor power as available. When there is little demand on the computational facilities of the system, much planning can be done. When there is no time to plan, no incremental planning need be done. Any incremental planning done will not be wasted when emergencies halt the planning process, for the results of the planning are immediately recorded.

Chapter 7

Analysis of Results

7.1 Introduction

The results given in this chapter are perhaps the most important part of this thesis. They represent approximately one computer year of run time, not including programming, debugging or analysis time. Results are given for the problem domains described in chapter 5: N-World, Octworld and Hexworld.

N-World was designed specifically to show off the advantages of planning and lookahead. There are really only three things to be learned in N-World. Because of this, the learning curve for N-World is more like a step function than a curve, and the analysis required is different from that of Octworld or Hexworld. Another simplification used in N-World was turning off the genetic algorithm. Instead, one classifier was created for each possible transition in N-World. These classifiers remain throughout the entire execution of each of the autonomous learners (*i.e.* the taxon and action are unchanged, while the other values associated with the various learning algorithms of the autonomous learners are updated as usual). Because of the specific nature of N-World, I consider success of lookahead classifier systems there to be more a validation of the concept than a demonstration of general utility.

Octworld and Hexworld are sufficiently general that to succeed there is an important test. The learning curves for these environments are initially steep, flattening out to some unknown asymptotic performance. A steep initial slope,

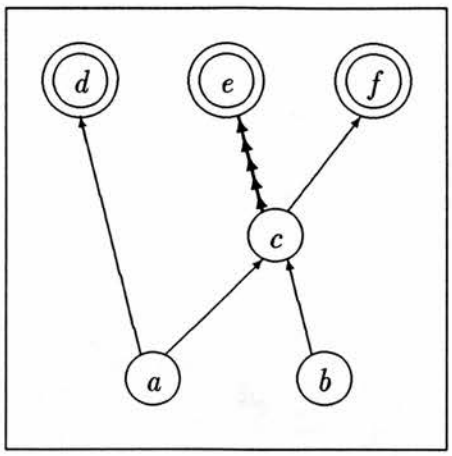

Figure 7–1: N-World.

indicative of rapid learning is especially important when the environment is variable, whereas asymptotic performance indicates the ability to more completely adapt when there is little change in the environment. The full genetic algorithms, as described in the relevant chapters (4 and 6), are used.

7.2 Validation of Concept – N-World

In order to validate the concept of lookahead classifier systems, I designed a simple virtual world, called N-World (see figure 7–1). The goal nodes of N-World are at d, e and f. The automaton is placed in node a or b (with equal probability). The automaton performs actions until it arrives at a goal node. The reward received at node f is 10000, arriving at node d gives a reward of 1000, and a zero reward is given for arriving at node e. The main difficulty in learning to maximize the reward obtained in N-World is that while most of the arrows in the diagram of N-World correspond to a single action, there are 496 actions in state c that result in a transition to state e. This almost guarantees that the single action that results in a transition from state c to state f will remain undiscovered for some time. The performance of the automaton improves as it learns the actions that move it to better paying goal nodes.

The performance of an automaton in N-World tends to make three major transitions. The first thing normally learned is the path from node a to node d, a moderately rewarded path. The second performance increase occurs when the path from node b to node f is learned, a highly rewarded path that is difficult to find. Finally, the path from a to f is found[1]. At this point the optimal paths are known, but not all paths have been explored (specifically there are many paths from c to e that are still likely to have remained untried). The performance continues to improve as these paths are tried, and discovered to be sub-optimal. In each of the 32 runs of the various automata through N-World the sequence of

---

[1]The path from b to f is normally discovered before the path from a to f for two reasons. First, even in a random walk model, node c will be reached half as often from node a as from node b, so paths out of c are investigated half as often when the automaton starts at a. Secondly, since the path from a to d is easily discovered, and rewards more than the path from a to c (before the discovery of the path from c to f), a learning automaton is even less likely to investigate paths out of c from a than would the random walk model.

Table 7–2: N-World simulation performance data

Algorithm	$\overline{x}_{ad}$	s_{ad}	$\overline{x}_{bf}$	s_{bf}	$\overline{x}_{af}$	s_{af}	n
Random Walk	4	2.0	992	31	1984	45	—
Reconstructed Animat	5	3.8	460	510	9279	2393	11
LA-Average	7	4.0	816	577	819	576	10
Q-CS	5	5.1	674	635	680	631	11

discovery was the same (although in ten of these runs, the path from a to f was never discovered).

The advantage of a topological model internal to the learning automaton is that once the connectivity and the values of each of the goals is known, the optimal paths can be determined by the automaton (either via lookahead, standard planning, or an incremental planning method such as that used in Dyna-Q-CS). Thus, in N-World, the path from a to f can be discovered by examination of the internal world model once the paths a to c and c to f are known. In the examples here, only lookahead was used to exploit the world model. N-World is sufficiently simple that the incremental planning scheme in Dyna-Q-CS provides no advantage beyond the lookahead already included in Q-CS. Because of this Dyna-Q-CS was not tested in N-World, and is represented by Q-CS[2].

Table 7–2 summarizes the results of the runs using the N-World simulation. The entry, $\overline{x}_{ad}$, is the mean time before the automaton discovers the path from a to d. The time is measured in iterations, where one iteration starts when the automaton is placed in node a or b of N-World and ends when the automaton reaches a goal node. The standard deviation from the mean for $\overline{x}_{ad}$ is in the

[2]In fact, a modified Q-CS was used in the N-World simulations. The intersection operation used in Q-CS was removed in order to increase the speed of the runs. The resulting algorithm only has a one move lookahead, rather than the two move look-ahead standard in Q-CS and Dyna-Q-CS. In the simple environment of N-World, the actions of this modified Q-CS will be the same as the unmodified version.

column headed s_{ad}. The mean times and standard deviations for the discovery of the paths from b to f and a to f are shown likewise. The value n is the number of sample runs represented by the statistics. The random walk values for these times are based upon theoretical considerations, and are Poisson distributions. The underlying distribution for the other algorithms is unknown, which makes comparative analysis between the different algorithms uncertain.

The statistic which is of most interest for the demonstration of the importance of lookahead is $\overline{x}_{af}$. When the path from a to c is known, and the path from (b to) c to f is known, lookahead should allow the the learning automaton to quickly discover the value of the path from a. Looking at the column labelled $\overline{x}_{af}$, this does indeed appear to be the case. Both La-Av and Q-CS show considerable improvement on the performance of the random walk and the reconstructed Animat[3]. The distributions would have to quite asymmetric for these differences not to be significant.

If we are not yet satisfied that lookahead has increased learning speed, there are two quantities we can examine that are more likely to be normally distributed and therefore more amenable to analysis: the difference between the discovery times of paths b to f and a to f, and the average reward per action for the total run. As these are composite quantities, they have a propensity towards being normally distributed[4].

---

[3]Ten out of the eleven runs of the reconstructed Animat in N-World failed to discover the path from a to f in the course of 10000 iterations. In these statistics, it is optimistically assumed that this path would have been discovered on iteration 10001.

[4]As the number of independent variables being summed approaches infinity, the central limit theorem guarantees that the composite value will be normally distributed. The more variables being summed, the more likely that the composite sum is normally distributed.

Formulae for Analyses

Given normal distributions for the underlying distributions, Student's t-test is appropriate. Two forms of Student's t-test are employed in these analyses, one for comparisons between the sample data and the theoretical random walk values, and a more complex form required for comparison between the empirical sample data associated with each algorithm [Walpole 1982].

If one is testing a sample against a known mean, μ_0 (such as the theoretical random walk values), the t value is:

$$t = \frac{\overline{x} - \mu_0}{s/\sqrt{n}}, \tag{7.1}$$

where $\overline{x}$ is the sample mean, s the square root of the sample variance, and n the number of sample points. The degrees of freedom are calculated:

$$v = n - 1. \tag{7.2}$$

Comparing two sample means is more difficult, especially when little is known about the underlying variance of the two distributions. The sample variance, s^2, is used as an estimator for the underlying variance, σ^2. The resulting equation is:

$$t' = \frac{(\overline{x}_1 - \overline{x}_2)}{\sqrt{(s_1^2/n_1) + (s_2^2/n_2)}}. \tag{7.3}$$

In order to get the degrees of freedom, v, the appropriate equation is:

$$v = \frac{((s_1^2/n_1) + (s_2^2/n_2))^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}. \tag{7.4}$$

The integration values for the t distribution are normally available in tabular form. The table I used assumed integer values for v, so I employed a floor function on the v value obtained.

Comparative Analyses

The mean time taken between the discovery of the path from a to f after the discovery of the path from b to f, along with the standard deviations, are listed under the headings $\overline{x}_d$ and s_d in table 7–3. Figure 7–4 is a significance diagram

Table 7–3: N-World simulation performance data

Algorithm	$\overline{x}_d$	s_d	$\overline{x}_{rew}$	s_{rew}	n
Random Walk	1984	45	255	—	—
Reconstructed Animat	8820	2342	3024	515	11
LA-Average	4.0	2.8	2624	92	10
Q-CS	6.2	6.5	4577	224	11

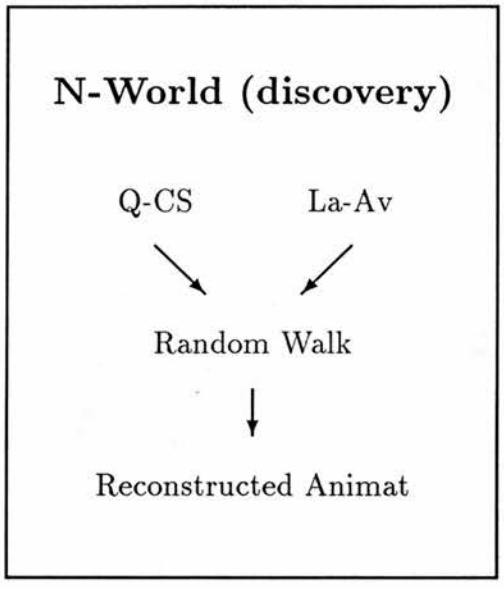

Figure 7–4: N-World — Discovery of optimum paths (significance diagram)

showing that La-Av and Q-CS do significantly outperform random walk[5], which significantly outperforms the reconstructed Animat. The results were significant at the $< .005$ level (*i.e.* there is $< .005$ chance that these results are due to random variation). This is the maximum significance discernible with the tables used.

---

[5]As I am looking for a causal link between these discoveries, I have only considered random walk solutions where the discovery of a to f follows discovery of b to f. If any of my experimental runs had shown reverse sequencing of these discoveries, I believe the run should have been left out of the comparison statistics.

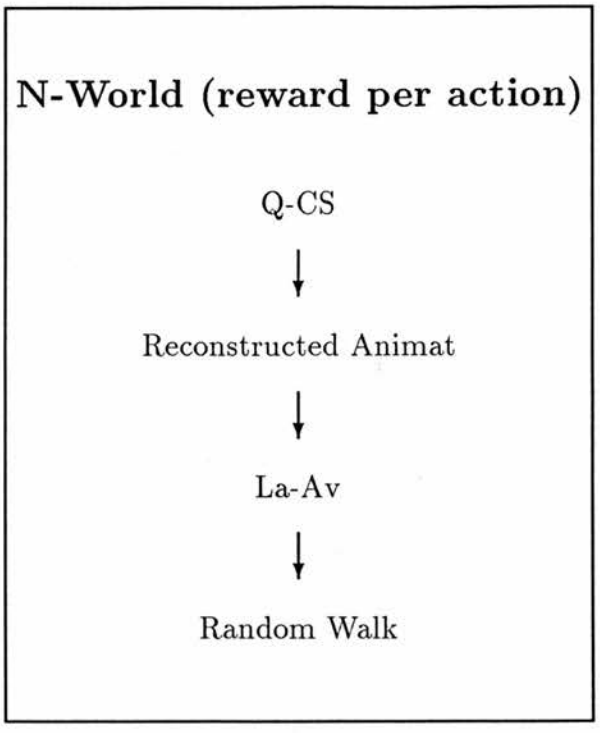

Figure 7–5: N-World — Mean reward per action (significance diagram)

Now that it has been established that La-Av and Q-CS are performing as designed in so far as finding new paths to well rewarded goals, let us see how this affects the overall performance of these algorithms. The entries $\overline{x}_{rew}$ and s_{rew} in table 7–3 show the mean reward per action for each algorithm and the standard deviation of this mean between different runs. The significance diagram (figure 7–5) shows the transitive relationship of Q-CS significantly outperforming the reconstructed Animat, which outperforms La-Av, which outperforms random walk. The only significance level worse than $< .005$ was the difference between the reconstructed Animat and La-Av which was at the $< .025$ level. La-Av is not performing nearly as well as might be expected. In order to understand how badly La-Av performs, it is instructive to take a closer look at the performance of the reconstructed Animat.

The single run of the reconstructed Animat that discovers the path a to f greatly increases the standard deviation of the sample. If we remove this data point from the sample, the reconstructed Animat *still* outperforms La-Av, at an

even higher significance level ($< .005$). So, we are left with an apparent paradox. La-Av, with the information about the existence of the optimum path from both start nodes, performs worse than the reconstructed Animat, even when the latter discovers only one of the two optimum routes. In order to understand how this can happen, it is useful to examine the graphs of some of these runs.

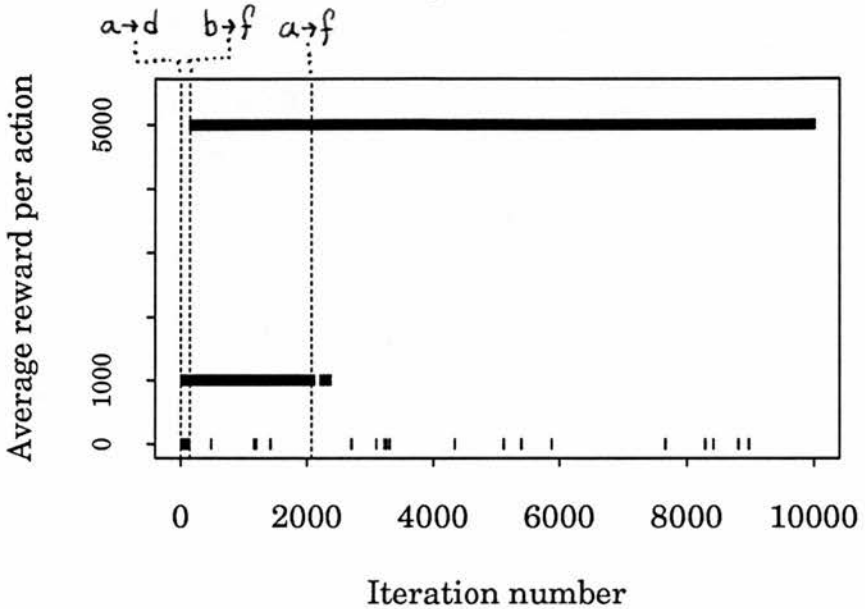

Figure 7–6: The Reconstructed Animat in N-World (best run)

The Reconstructed Animat

Figure 7–6 is a graph of the best run of the reconstructed Animat in N-World. It is the run that found the path from a to f. For each iteration, there is a vertical tic mark showing the amount of reward divided by the number of actions required to obtain the reward. The appearance that the graph is multi-valued is an illusion. Nearby tic marks coalesce due to insufficient resolution. In places where the graph appears to be multi-valued, it is merely an area where the average reward per action is rapidly fluctuating. The vertical dotted lines represent the points (from left to right) at which the paths a to d, b to f, and a to f were discovered by the reconstructed Animat. The graph is much as one might expect. The path from a to d is learned very quickly. Starting from a, it finishes at d. Starting from b it finishes at e. This causes the solid marks at zero and 1000. Once the path from b to f is discovered, zero rewards are seldom encountered. When the path from a to f is learned, visits to d, and 1000 rewards quickly cease. The graphs of other

runs of the reconstructed Animat are much the same, except that the discovery of the path a to f does not occur, so the changes this causes to the graph are eliminated. Once the reconstructed Animat makes a discovery, it continues to exploit this discovery from then on.

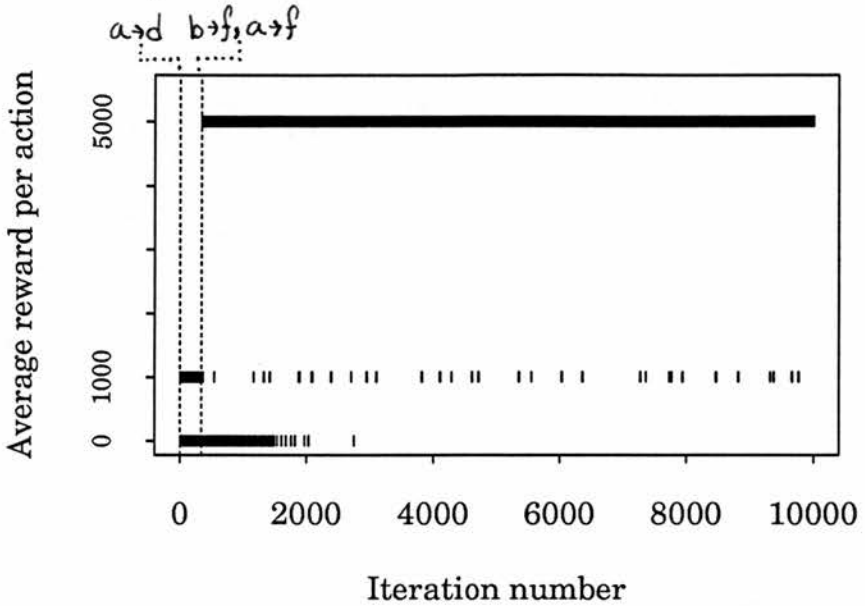

Figure 7–7: Q-CS in N-World (run selected at random)

Q-CS

The graph for Q-CS (see figure 7–7) is rather different from that of the reconstructed Animat. While the graphs look much the same from the first iteration up until the discovery of the path from b to f, to the right of it things are quite dissimilar. It does not show up well on the graph because of the resolution, but the path from a to f is found immediately after the discovery of the path from b to f. Thus, the two discoveries appear as a single dotted line. Even after these discoveries, Q-CS continues to investigate untried paths from c to e. While in N-World this turns out to be to no avail, one could easily construct worlds where this investigation was rewarded. The reconstructed Animat does not tend to do much of this sort of investigation.

Other runs of Q-CS in N-World were similar to the one shown. The runs are revealing in that the visits to node d continue, even in the presence of a path to the more highly rewarding node f. This is a consequence of how Q-CS chooses actions. Node d will continue to be visited a fixed ratio of the time. This is not

desirable. Sutton [1990] and Watkins [1989] use a form of simulated annealing to avoid this in Q-learning systems. Adapting these mechanisms to Q-CS is discussed in chapter 8.

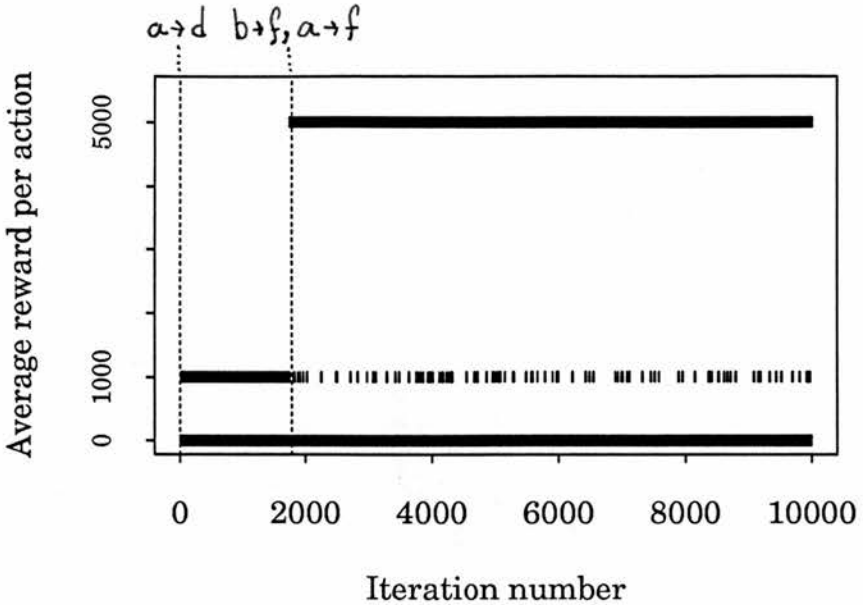

Figure 7–8: La-Av in N-World (run selected at random)

La-Average

As shown in figure 7–8, La-Av shows the now familiar pattern of quick discovery of the path from a to d, followed by exploitation of this discovery. Soon, the path from b to f is discovered[6]. As with Q-CS, the discovery of the path from b to f quickly triggers the discovery of the path from a to f. From that point onward the graph is quite different. While the highly rewarded goal at node f is visited often, the unrewarding goal at node e is visited frequently as well. The medium reward at d continues to be investigated at intervals. This strategy does not produce optimal results in N-World, and requires some explanation.

---

[6]The particular run shown in the figure discovers this path relatively late. If one assumes an underlying Gaussian distribution for the items in table 7–2, there is only suggestive ($< .1$ significance level) evidence that the mean time for discovery of the path from b to f is worse for La-Av than for the reconstructed Animat. Suggestive evidence based upon an uncertain assumption is hardly convincing. One should, therefore, not take the late discovery of the path from b to f in figure 7–8 too seriously.

N-World was designed to be easy for lookahead systems like La-Av. La-Av quickly learns the optimal paths, but does not exploit them. The reason for this is the way La-Av handles the unknown. As explained in chapter 6, each classifier has an associated bag of situations that have followed execution of that classifier. The number used in determining the probability that a classifier will be selected is an average of the value of all the situations in this list, along with an estimator of the value of the unknown. In La-Av, this estimator is the average rating (*strength*/*distance*) of every classifier in the system. While this average may be fairly small, there are 496 classifiers making use of that rating to suggest an action that will move from c to e. Since each of these classifiers has a separate action, the effect will be cumulative against the single classifier specifying the move from c to f. It may take a long time before the experience of the automaton finally overwhelms this summation.

Conclusion

While N-World is a very simple place, it has served well to distinguish the performance of the different algorithms. The reconstructed Animat is very conservative. The exploitation versus exploration problem has been solved in favour of exploitation. In Q-CS and La-Av, there is more exploration. The use of lookahead has been shown to be valuable in discovering useful paths. Both Q-CS and La-Av benefited from this ability. Q-CS was very successful, but the asymptotic performance is limited due to the strategy for selecting actions. La-Av did not perform well in N-World, because its estimate of the unknown was not well suited to N-World. La-Av explores and re-evaluates places it has already visited. In an ambiguous world, this re-evaluation could prove useful.

In worlds where the value of a node varies over time, for example, the re-evaluation strategy of La-Av might be more suitable. La-Av takes a long time to model the value of a node, and its time would be well spent. The reconstruction of Wilson's Animat is quickly attracted to what it detects as high payoff nodes. Once sufficiently high payoff is reached, further investigation ceases. I expect that the reconstructed Animat would choose constant rewarding nodes, rather than high

paying, but variable valued nodes. Q-CS is sufficiently methodical to search the space of actions more thoroughly than the reconstructed Animat, but a node that initially showed poor reward value might not be further investigated.

While some strategies are inherently superior to others, it is often the case that the best strategy is dependent more specifically on the problem being considered. In such cases it is useful to find out exactly the areas in which particular strategies are strong or weak. This investigation continues in Octworld and Hexworld.

7.3 Octworld and Hexworld

N-World is a very simple place. There is no possibility of getting caught in a loop. The nodes are unambiguously labelled. The task of the classifier system was simpler as well. The classifiers were hand constructed and the genetic algorithm was suppressed. In Octworld and Hexworld, things are more complicated. There are many more nodes, and the information available at each node is not sufficient to distinguish the node. It is possible to loop indefinitely in these worlds. All classifiers are constructed probabilistically, or via genetic algorithm. New classifiers are introduced, and extant classifiers are deleted.

Octworld and Hexworld have been described more fully in chapter 5. For each algorithm tested data was collected by repeatedly placing the automaton in an unoccupied node of the world. The number of actions required to reach a goal node was recorded. A full run consisted of 1000 such events, with the automaton allowed to improve performance as experience was gained. The number of moves required was plotted against iteration number. This produces a curve with decreasing values as the automaton learns. This is the opposite from the technique used to display learning curves in N-World, where learning produced higher values[7]. When this data is viewed, the shape of the learning curve appears to be roughly an exponential decay, with the y-intercept at the random walk

[7]Since the rewards given in Octworld and Hexworld are all the same value (1000), one could derive a curve more compatible with those of N-World (specifically, by plotting

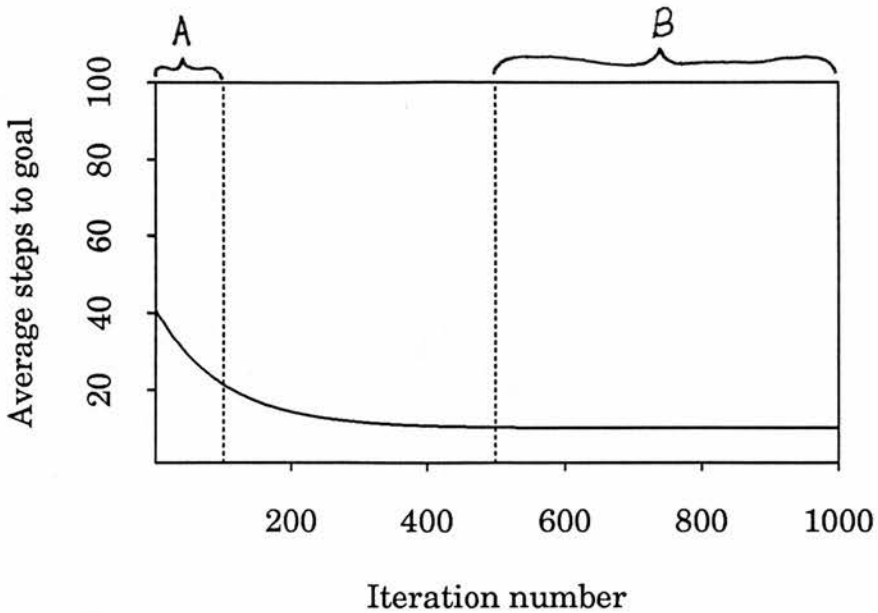

Figure 7–9: A theoretical learning curve

performance level. From there the curve drops steeply at first as the iteration numbers increase, and then the curve flattens out approaching its asymptote. Figure 7–9 shows an idealized diagram of the general shape of these curves. There is a very large amount of noise in the actual data, as will become obvious later. I was unable to determine the precise form of the curve, so I have not attempted a parameterization. Instead, the curve was analyzed at two crucial segments: the initial steep decline (henceforth phase A) and the flatter slow decline later on in the learning curve (phase B). For the purposes of the numerical analyses presented here, the values of the first hundred iterations were used as phase A, and the values of the last five hundred iterations were used for phase B. The performance of each algorithm over the entire run was also analyzed.

1000 divided by the value shown in these graphs). The curves as currently plotted are more compatible with Wilson's original work.

Analysis Techniques

For each of the curve segments analyzed, two methods of comparative analysis were used. They both have their strengths and weaknesses, and it is hoped that the use of both will provide greater insight into the performance of each algorithm. The equations used in both techniques are derived from *Introduction to Statistics* [Walpole 1982]. They are both t-test analyses. Significance values were obtained from a table with entries for .1, .05, .025, .01 and .005 level of significance.

The first technique, which I will call the "distribution" technique, involved taking the average value of the curve over the segment being analyzed. For each algorithm being analyzed, these average values form a distribution, each with a mean and standard deviation. This is basically the same technique used previously to analyze the average reward per action for runs in N-World. Re-iterating equations 7.3 and 7.4, the Student t-test value for the comparison is:

$$t' = \frac{(\overline{x}_1 - \overline{x}_2)}{\sqrt{(s_1^2/n_1) + (s_2^2/n_2)}},$$

and the degrees of freedom, v, are calculated:

$$v = \frac{((s_1^2/n_1) + (s_2^2/n_2))^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}.$$

As before, I used the floor of the v value in order to obtain significance figures from a table.

The t-test is only valid if the underlying distribution is normal. As the data is the result of summing numerous data points this is a reasonable assumption (due to the central limit theorem). The reasonability of this assumption is further strengthened by noting that the sample standard deviation of the data points is much smaller than the sample mean. If this was not the case, it would be difficult to account for the fact that no data points can ever be negative.

A second analysis technique was also used. One problem with the preceding analysis technique is a lack of sensitivity. The data points being analyzed tend to improve as the iteration number increases. The previous analysis views this as mere noise, and the standard deviation becomes too large to produce significant results. In order to minimize this problem, another technique, which I will call

the "difference" technique, based upon iteration number, was devised. For each algorithm, an average learning curve was produced where the value for iteration n was the average of the n-th iteration value for each individual run. To compare the performance of two algorithms, a distribution is formed from the differences of the corresponding elements of the averaged learning curves. This should produce a normal distribution (again, because of the central limit theorem). Whether one algorithm performs better than the other is equivalent to whether this new distribution has a mean significantly different from zero. If the mean is greater than zero, then the first algorithm has higher values on average than the second algorithm. If the mean of the distribution is negative, then the converse is true. The significance test for the distribution, d, is again a t-test (adapted from equation 7.1):

$$t = \frac{\overline{d}}{s_d/\sqrt{n}},$$

and the degrees of freedom are calculated:

$$v = n - 1.$$

The main problem with this analysis is a possible bias. Since the number of individual runs available for each algorithm is small, the averaged learning curve may be far from the true values. Before continuing the analysis, it may be useful to review the worlds that we are dealing with.

Octworld

Octworld has been fully described in chapter 5. It is an eight-connected graph with clusters of associated obstacles and goals. From a global perspective, it is a toroidal surface. Octworld is the virtual world used by Wilson in his Animat work.

The learning curves for the reconstructed Animat, La-Av, Q-CS and Dyna-Q-CS in Octworld are shown in figures 7–10, 7–11, 7–12 and 7–13, respectively. These are the averaged curves as used in the "difference" analysis. Since different numbers of runs were used to create the different graphs, they show more or less noise in the curve (the more runs averaged, the less noise in the curve). Some of

these curves appear quite flat (other than the considerable noise spikes). It should be pointed out, therefore, that all of the algorithms do better than random walk. As the classifiers in these systems are created with random values, and should start out roughly as random walk routines, learning has taken place in all these curves. If a curve looks flat, it is indicative of a steep slope of learning very early in the history of the runs. This is the noisiest portion of the data, so the shape of this steep slope remains hidden.

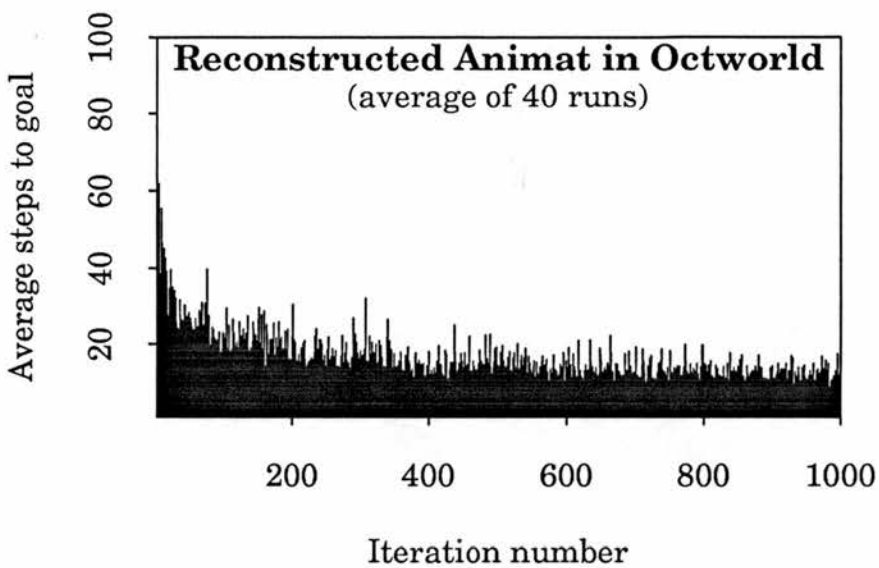

Figure 7–10: The Reconstructed Animat in Octworld

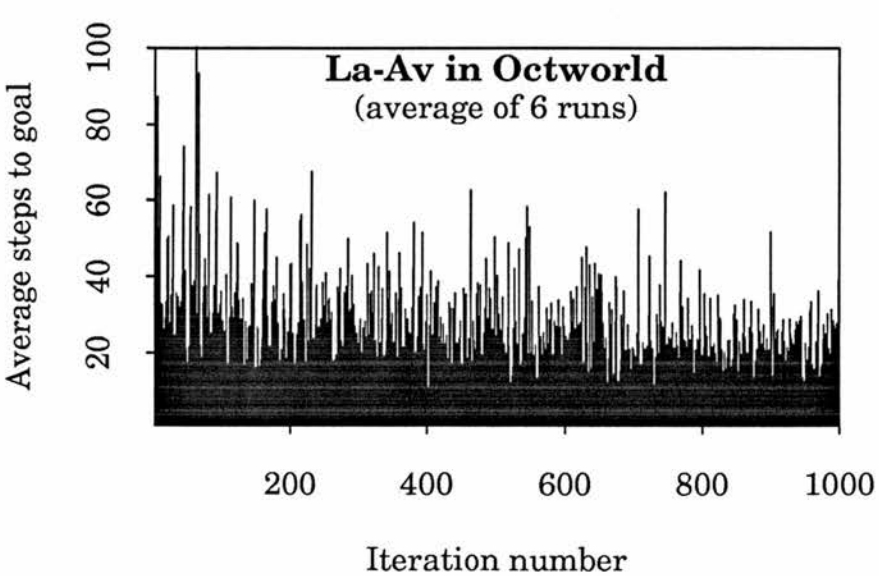

Figure 7–11: La-Av in Octworld

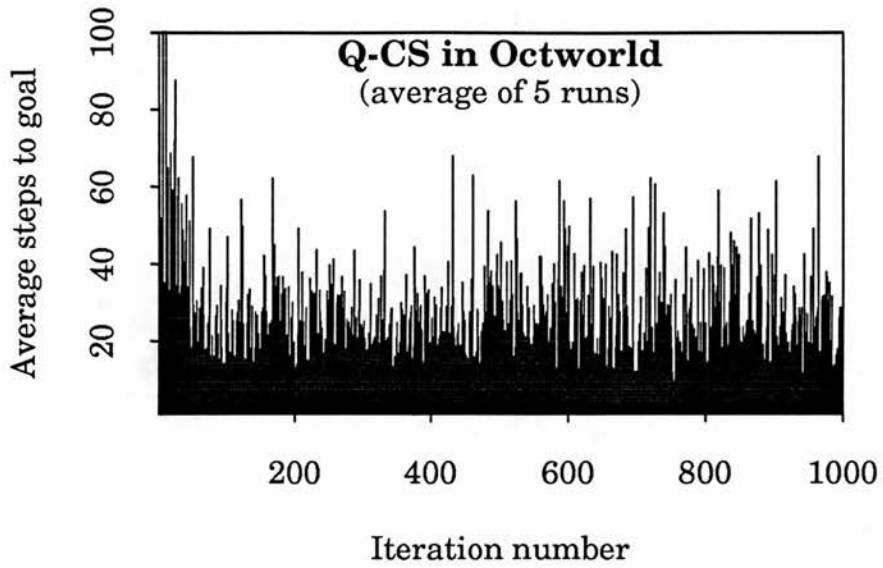

Figure 7–12: Q-CS in Octworld

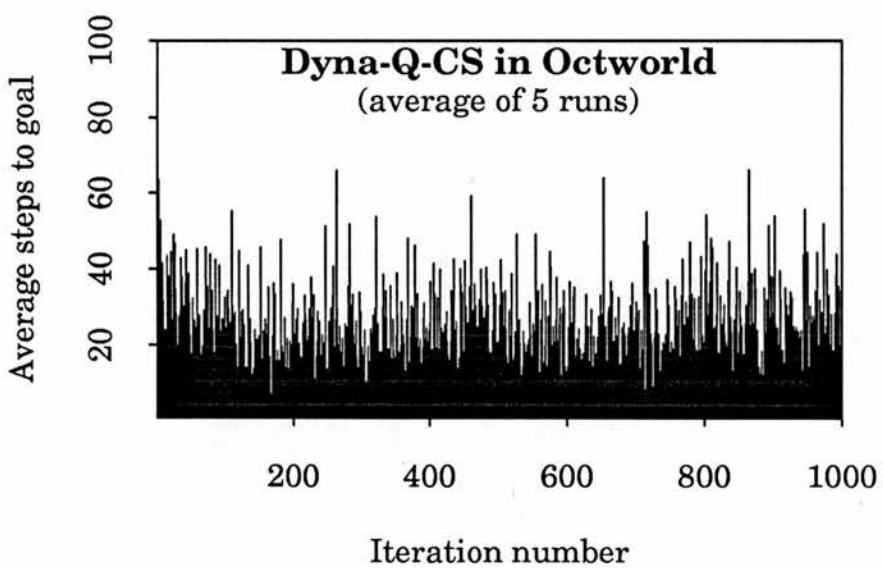

Figure 7–13: Dyna-Q-CS in Octworld

Hexworld

Hexworld is a six connected variant of Octworld. It can be viewed as Octworld where the two directions, Northeast and Southwest, are unavailable. Topologically this is now a hexagonal tiling of the surface of a torus. Since two directions are now unavailable, some nodes of the graph are further apart than in Octworld. Less information is available to the automaton.

To offset these disadvantages of topology, there are two factors that may make Hexworld an easier place to learn about. All the directions are symmetrical, so there are no favoured directions. Probably of even more importance is that there are fewer possible classifiers in Hexworld by a factor of 108 (the taxon is four elements shorter, with three possible values for elements, and there are six rather than eight directions, making the equation $3^4 \times 8/6 = 108$), so searching through the classifier space should be much quicker.

The learning curves for the reconstructed Animat, La-Av, Q-CS and Dyna-Q-CS in Hexworld are shown in figures 7–14, 7–15, 7–16 and 7–17, respectively. As with the graphs presented for Octworld, these are the averaged curves as used in the "difference" analysis.

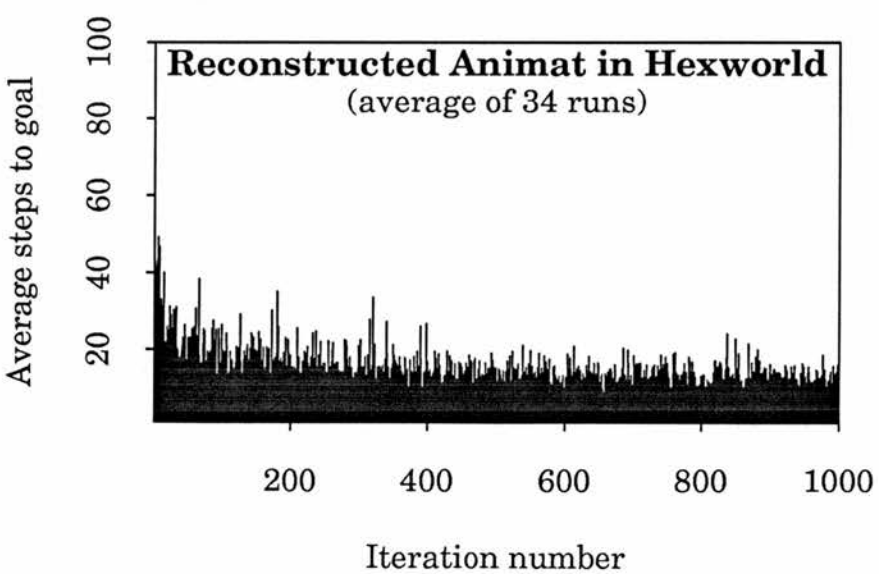

Figure 7–14: The Reconstructed Animat in Hexworld

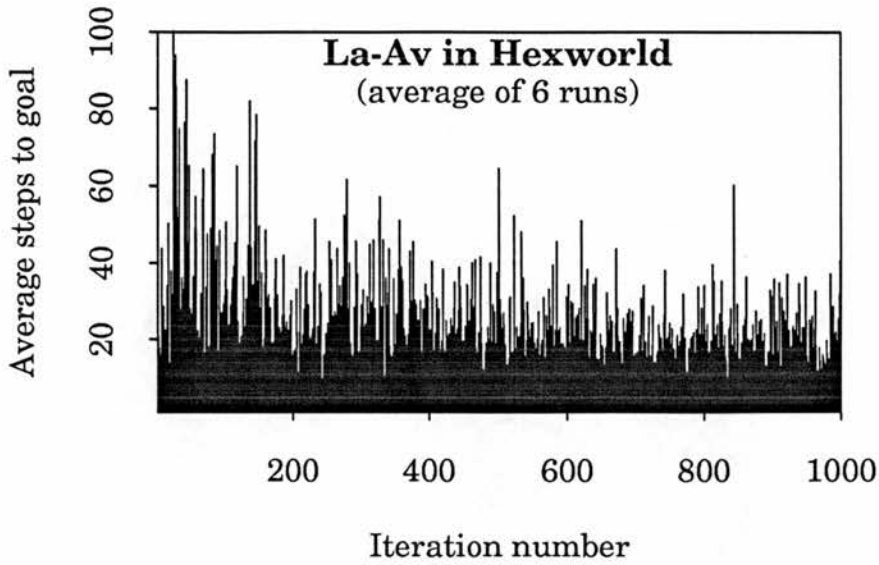

Figure 7–15: La-Av in Hexworld

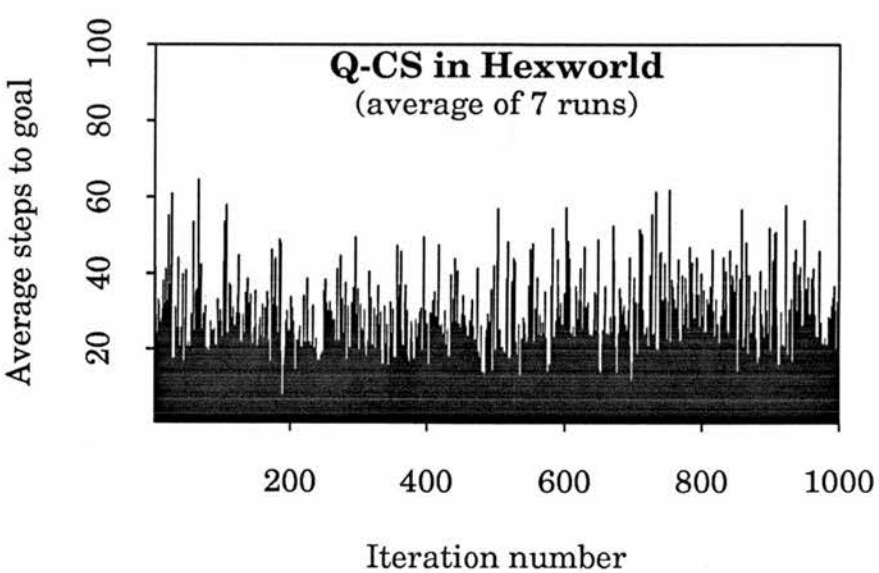

Figure 7–16: Q-CS in Hexworld

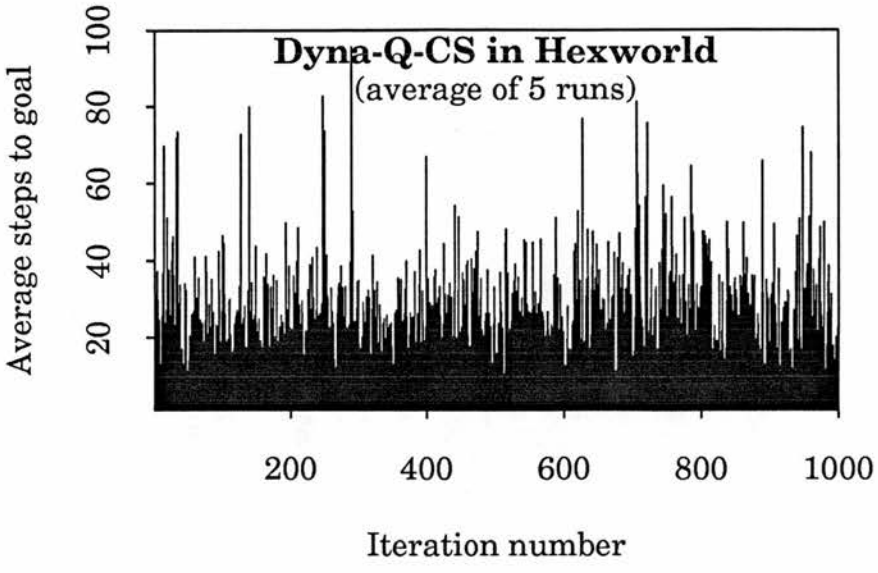

Figure 7–17: Dyna-Q-CS in Hexworld

7.3.1 Analysis and Comparison of Results

The information in the following tables is that required for the "distribution" method of analysis. These tables are provided for insight into the actual numbers we are dealing with. They include the sample mean, $\overline{x}$, sample standard deviation, s, and number of runs over which this data was collected, n. The figures are based upon both the "distribution" and the "difference" methods of analysis. An arrow from one algorithm to another means that the former is significantly (.05 or less chance of this being a random effect) better than the latter. The arrow relation is transitive. Arrows labelled with an asterisk, "*", are only significant under the "difference" method of analysis. This does not affect the interpretation of a transitive relationship. If there is a transitive relationship between any two algorithms, the first is significantly better than the last under both analyses. Any arrow without significance $< .005$ by either of the analysis methods will have its significance mentioned in the text. For example, in figure 7–19, one can tell that under the "distribution" analysis, the reconstructed Animat performed significantly better than all three of the algorithms with world models. Under the "difference" analysis this is also true, but further, Dyna-Q-CS performed significantly better than either Q-CS or La-Av. It is left to the text to mention any significance not at

Table 7–18: Octworld — Iterations 1 – 1000 ("distribution" data)

Algorithm	$\overline{x}$	s	n
Reconstructed Animat	15.1424	3.313	40
LA-Average	23.3555	4.983	6
Q-CS	23.294	2.066	5
Dyna-Q-CS	21.8048	1.818	5

the $< .005$ level, including suggestive ($< .1$) results that will not be shown in the figure.

Results for the Total Run

Table 7–18 is an analysis, for Octworld, of each algorithm over the entire run. While genetic algorithms are meant to optimize performance over the entire run, viewing these results as the most important can be misleading. A fast learning algorithm excels at the start, yet it is the asymptotic performance that will eventually dominate. Which factor dominates is merely a function of the length of the run.

Figure 7–19 shows that the lookahead techniques fall significantly short of Wilson's method. Using the "distribution" technique, the preeminence of Dyna-Q-CS over the other look-ahead systems is not noticeable, but the overall dominance of the Wilson reconstruction is obvious (although under the "distribution" analysis, the significance of the Reconstructed Animat over La-Av is at $< .01$).

From the data for Hexworld in table 7–20 a somewhat different picture emerges (figure 7–21). While the reconstructed Animat again dominates, the bucket brigade algorithms both outperform the Q-learning systems. Under the "distribution" analysis, La-Av is better than Q-CS at the $< .05$ level of significance, and there is a suggestive probability that La-Av is also better than Dyna-Q-CS. This is compatible with the conclusion reached by the "difference" method.

The cause of the difference between the results is of interest. The evidence from these runs shows that in the first hundred iterations, the Hexworld performance

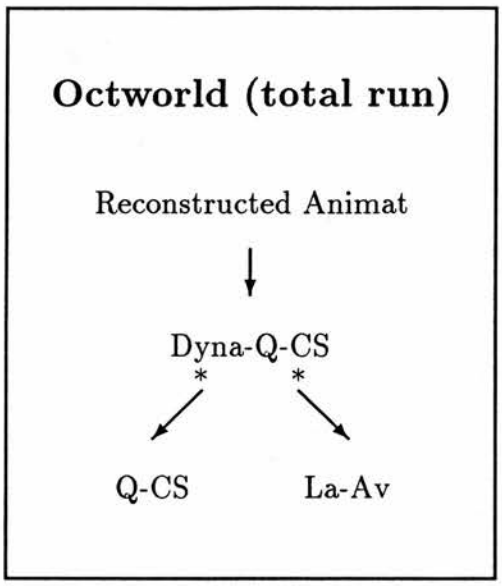

Figure 7–19: Octworld — Iterations 1 - 1000 (significance diagram)

Table 7–20: Hexworld — Iterations 1 - 1000 ("distribution data")

Algorithm	$\overline{x}$	s	n
Reconstructed Animat	14.79168	2.820	34
LA-Average	21.92867	1.781	6
Q-CS	24.22086	2.277	7
Dyna-Q-CS	24.5836	2.753	5

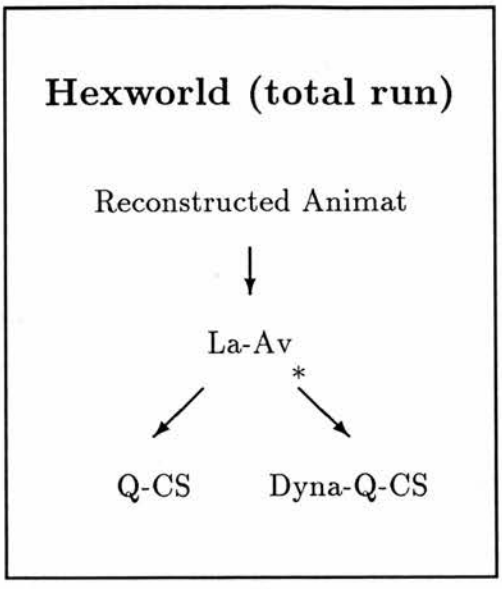

Figure 7–21: Hexworld — Iterations 1 – 1000 (significance diagram)

is better than that in Octworld, even though Hexworld intrinsically requires more moves to reach a goal. I believe this has to do with the fact that the initial population of 500 classifiers is a larger percentage (by a factor of 108, as calculated earlier in this chapter) of the possible classifiers in Hexworld than in Octworld. Better initial performance suggests a steeper initial learning curve (phase A) which must flatten to phase B, and asymptotic performance, earlier. Because of this, whereas the Octworld results are dominated by phase A results, the Hexworld results are dominated by phase B. This hypothesis shall be examined further as the analysis continues.

Asymptotic and Phase B Performance

Asymptotic performance is a very important measurement. It shows how well, given much time and experience, an algorithm can optimize its performance. Unfortunately, asymptotic performance is, except in degenerate cases, never reached. Without an equation for the curves in question, asymptotic performance cannot be deduced. Therefore, rather than studying asymptotic performance, we can only look towards the end of the data we have, and analyze that. This is the phase B performance, the last 500 iterations of our test runs. The learning curves have

flattened quite a bit by that time, and much that might be said about asymptotic performance can be said about phase B performance. In both asymptotic and phase B performance, much information is known about the world. There is a tradeoff, however, between optimal performance and investigative ability. If the world has truly been investigated and is static, optimal performance could be calculated and obtained, but as I have argued earlier, there is no way to tell when a world has been fully investigated. Therefore there is always the motivation to try an action that appears sub-optimal, in order to further investigate the world, which may allow even better performance in the future. Further, we do not necessarily want to optimize for static conditions.

Table 7–22 includes some very good performers. The theoretical optimum[8] is not obtainable by any automaton with the sparse sensing capabilities allowed in Octworld. The "hand crafted automaton" is a classifier system based upon the reconstructed Animat. I designed classifiers intended to perform well in Octworld, and turned off both the genetic algorithm and bucket brigade learning schemes. Duplicating this performance with a learning automaton seems a reasonable goal. Wilson's results [Wilson 1985] are for an automaton that has been trained for 8000 iterations. The "trained reconstruction" entry is for my reconstructed Animat, trained for 9000 iterations. The figures are for the averages over the next 1000 iterations[9]. This "trained reconstruction" shows the folly in considering Phase B values to be asymptotic values, since the reconstructed Animat clearly continues to learn well after the simulations reported in this chapter have completed. However, there is still some reason for confidence that the comparative phase B values reflect the *relative* asymptotic performance of these algorithms[10].

---

[8]The values listed in the table are actual distribution values, not sample values

[9]For more information on this, and a comparison between Wilson's Animat and the reconstructed Animat, see chapter 4.

[10]For example, if a logarithmic analysis for the shape of the learning curves is correct, there should be at most two points where two curves intersect. Since all of the systems start roughly as a random walk, the curves should all intersect at iteration zero. If

I will not dwell on a detailed comparison of the many entries in table 7–22. For any entry in that table not reflected in figure 7–23, any difference between two means should be considered significant. That is, the theoretical optimum is significantly better than the performance of the hand crafted automaton, which is better than that of Wilson's Animat, which is better than that of the trained reconstruction. The performance of these algorithms is, in turn, better than the performance of all the other algorithms listed in the table. Random walk[11] is significantly outperformed by every other algorithm in the table. All significances are at the $< .005$ level. If one is attempting to compare the performance of various algorithms, however, it makes sense to only compare them with the same amount of training. Wilson's Animat and the trained reconstruction are much further along their respective learning curves than the other learning automata[12]. Such figures are mainly included for reference.

Consistent with my hypothesis that Hexworld automata converge towards asymptotic results sooner than Octworld automata, the Hexworld picture (figure 7–25) is much more complete than that of Octworld (figure 7–23), even though the Phase B performance in Hexworld is generally worse (with the notable exception of the performance of La-Av—See tables 7–22 and 7–24). There is suggestive evidence from the "difference" analysis that the phase B Octworld performance of La-Av is better than that of Dyna-Q-CS, which would make the two pictures

---

there is another point of intersection, it will be definitive. At iterations between that point and zero, one curve will be superior, and beyond that point, the other curve will be superior. Evidence exists from the data presented in this chapter for such an intersection point between Dyna-Q-CS and La-Av. So the asymptotic relation between these two algorithms at least seems established.

[11]The random walk value in table 7–22 is a sample mean reported by Wilson [1985].

[12]A visual estimate for the phase B performance of Wilson's Animat can be gained from one of the charts in his article [Wilson 1985]. I put it somewhere under 6, and significantly better than the phase B performance of the reconstructed Animat, La-Av, Q-CS, or Dyna-Q-CS.

Table 7–22: Octworld — Iterations 501 - 1000 ("distribution data")

Algorithm	$\overline{x}$	s	n
Theoretical Optimum	2.184	0.868	—
Hand Crafted Automaton	3.1	—	~ 50
Wilson's Animat	4 - 5	—	—
Trained Reconstruction	6.65	1.08	41
Reconstructed Animat	12.1929	3.022	40
LA-Average	21.574	3.552	5
Q-CS	22.6552	2.531	5
Dyna-Q-CS	21.7032	3.503	5
Random Walk	41	—	—

almost identical. However, the "difference" analysis also provides suggestive evidence that the phase B Octworld performance of Dyna-Q-CS is better than that of Q-CS. I can see no theoretical justification for this to be true of the asymptotic performance, so I am skeptical that it is true of phase B performance[13]. More importantly for my convergence hypothesis, the total run figure for Hexworld (7–21) is virtually identical to the Hexworld phase B figure (7–25).

Learning Speed and Phase A Performance

The speed at which an algorithm can successfully adapt to its environment is crucial, especially if the environment is rapidly changing. Since all the algorithms being investigated start off roughly as a random walk, the performance of the automaton over the first hundred iterations is an indication of how fast it has learned. This is the steep phase A of the learning curve.

The key point demonstrated by the N-World tests is the utility of a world model to more quickly exploit the data it has available. This is the area in which we would

[13]There are several possible explanations as to why the suggestive evidence might be true, but I remain skeptical.

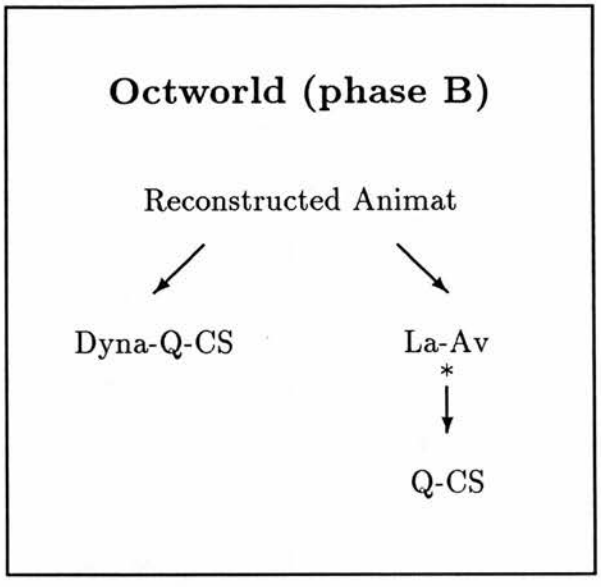

Figure 7–23: Octworld — Iterations 501 – 1000 (significance diagram)

Table 7–24: Hexworld — Iterations 501 – 1000 ("distribution" data)

Algorithm	$\overline{x}$	s	n
Reconstructed Animat	12.7321	2.828	34
LA-Average	18.5777	3.141	6
Q-CS	25.558	3.739	7
Dyna-Q-CS	25.3484	1.88	5

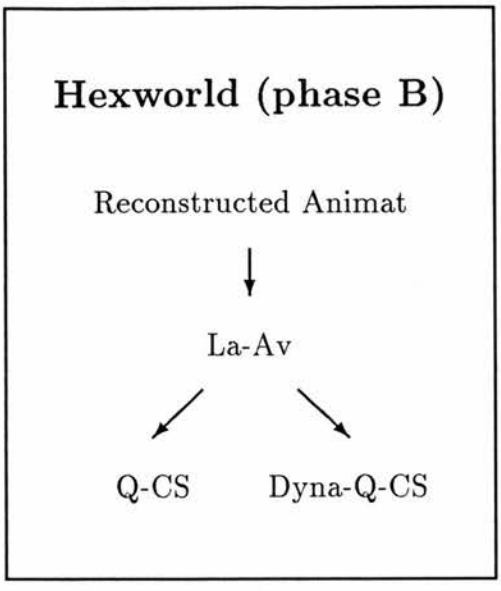

Figure 7–25: Hexworld — Iterations 501 - 1000 (significance diagram)

expect the lookahead systems and dynamic planning to excel. Tables 7–26 and 7–28, and figures 7–27 and 7–29 indicate that this is not particularly the case. While the sample mean value for Dyna-Q-CS in Octworld is lower than that of the reconstructed Animat, the difference is not significant. The performance of La-Av and Q-CS in Octworld is significantly worse. The first part of the learning curve is the most noisy, so significant results are harder to produce (especially for the "distribution" method which treats the slope of the curve as noise). In Octworld, the superiority of the reconstructed Animat and Dyna-Q-CS over La-Av is at the $< .05$ level of significance under the "distribution" analysis, and their superiority over Q-CS under the "difference" analysis is at the $< .025$ level of significance. In Hexworld, under the "difference" analysis, the superiority of Q-CS over La-Av is at the $< .01$ level, and there is suggestive evidence that the reconstructed Animat is superior to Q-CS. Under the "distribution" analysis the superiority of the reconstructed Animat over Q-CS is at the $< .025$ level, Dyna-Q-CS is superior to La-Av at the $< .05$ level, and there is suggestive evidence that the reconstructed Animat is better than Q-CS, which in turn is better than La-Av. The phase *A* figures, along with the suggestive evidence, are consistent with each other and with the hierarchy from complete runs of Octworld. This provides the final support for

Table 7–26: Octworld — Iterations 1 – 100 ("distribution" data)

Algorithm	$\overline{x}$	s	n
Reconstructed Animat	27.08876	8.000	40
LA-Average	33.64834	7.157	6
Q-CS	31.52	7.367	5
Dyna-Q-CS	26.634	1.245	5

my hypothesis that the difference between the Octworld and Hexworld full run hierarchies, figures 7–19 and 7–21, is caused by faster convergence towards the asymptotic performance in Hexworld than in Octworld.

The choice of the first hundred iterations for phase A, rather than the first fifty, or the first two hundred was fairly arbitrary. Too large a number would bias the result towards the asymptotic performance, and too small a number would give too much noise to produce significant results. I chose a point roughly where most of the Octworld curves started to flatten out. Unfortunately, the place where the curves start to flatten out is dependent upon the learning speed. I suspect that particularly in the case of Dyna-Q-CS in Octworld, the number of iterations chosen biased the result too far towards the asymptote. The reasoning behind my claim is the very low standard deviation of the distribution of averages shown in table 7–26. Such a low standard deviation suggests that Dyna-Q-CS has already settled down. It appears that Dyna-Q-CS learns extremely quickly, but, as expected, suffers from the same poor asymptotic performance as Q-CS.

7.4 Conclusion

The N-World simulations have served well to distinguish the reconstructed Animat, Q-CS and La-Av. The reconstructed Animat is a conservative and exploitative algorithm. It does little investigation, but does very well with the information it has. Q-CS is investigative, and exploits the information it has fairly well. There are problems, however, with its asymptotic performance. La-Av is quick at making

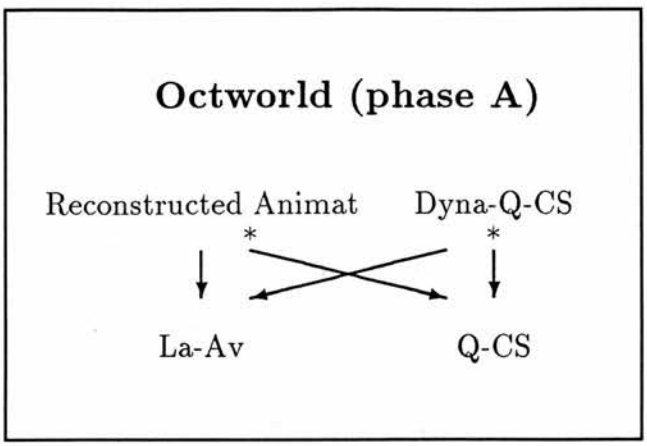

Figure 7–27: Octworld — Iterations 1 – 100 (significance diagram)

Table 7–28: Hexworld — Iterations 1 – 100 ("distribution" data)

Algorithm	$\overline{x}$	s	n
Reconstructed Animat	22.93882	4.996	34
LA-Average	32.605	9.232	6
Q-CS	24.98428	2.370	7
Dyna-Q-CS	24.292	3.815	5

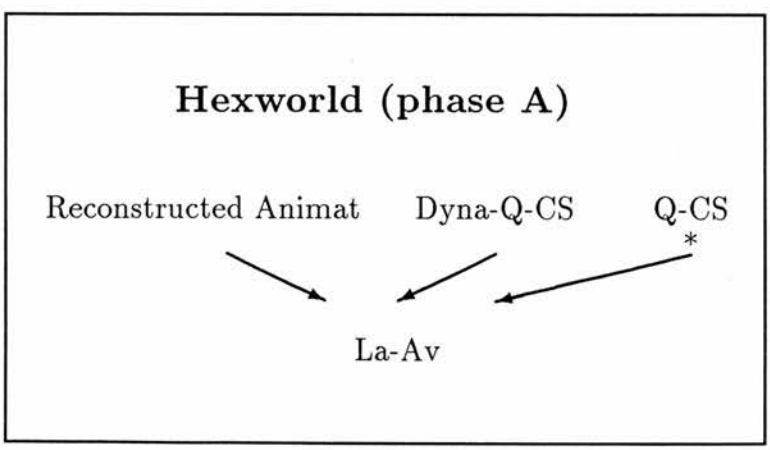

Figure 7–29: Hexworld — Iterations 1 – 100 (significance diagram)

discoveries, but does not exploit the discoveries it makes. Its evaluation of the unknown could use some attention. It is largely the simplicity of N-World that allows this analysis. In a more complex environment, it is more difficult to find the causes of performance problems.

In Octworld and Hexworld, the systems with lookahead did significantly worse than the reconstructed Animat. There are clearly some factors in Octworld and Hexworld that not only annul, but completely overturn the advantages of lookahead as shown by Q-CS in N-World. This still leaves the challenge of determining what these factors are, and seeing whether they can be more successfully handled while maintaining the clear advantages shown by lookahead systems in worlds like N-World.

The research demonstrates that Q-CS learning is feasible, although there are still adjustments to be made. Dynamic planning does appear to speed the learning process. There are numerous improvements that have yet to be borrowed from Q-learning. The interaction between genetic algorithms and classifiers systems with lookahead needs to be better researched. Such matters are discussed in the following chapter.

Chapter 8

Future Research Paths

8.1 Introduction

Given the various constraints upon ones research, there are always things one would have wished to accomplish. In fact, it would be poor research if it did not suggest new ideas to be tried and experiments to be performed. In the case of my research, there are various improvements that might be made to the algorithms. I would also have preferred to be able to use a greater variety of experimental worlds.

8.2 Genetic Algorithms and Classifier Systems

There is not all that much written on the inter-relationship of genetic algorithms and classifier systems[1]. In particular, when classifier systems are used as predictive models, the value of a classifier from a genetic standpoint and from the standpoint of actually wanting to activate a classifier are quite different. A classifier which predicts the occurrence of the undesirable is still useful to the classifier system. It should be propagated by the genetic algorithm, but rarely activated by the

[1]The best article I have seen on the topic is by Booker [1985], which is a summary from his thesis research [Booker 1982]

classifier system. It appears that some measure of reliability is more important from the genetic standpoint than the strength value of the classifier. It might even be desirable to mutate unsatisfactory predictors to have a more specific taxon (sensitive to fewer situations) in order to improve its predictive ability.

A more general problem has to do with the global nature of the competition being waged by classifiers. It does not make sense for classifiers that are close to a goal, to wipe out, via the genetic algorithm, the very classifiers which move the automaton to the position where these can be used. The classifiers further from a goal tend to have less strength (via the tax system in Bucket brigade, via the very nature of discounted future reward in a Q-learning type system). It would make more sense for classifiers that compete phenotypically, that is, they often compete to have the automaton take different actions, to also compete genotypically. One way to accomplish this would be to hold genetic competitions every time an action is determined, with parents and deletions occurring only within the set of classifiers with satisfied taxons.

8.3 Riolo's CFSC2

It is in the nature of research that different researchers will independently investigate similar areas. Rick Riolo [1990] has investigated lookahead within classifier systems with CFSC2. CFSC2 makes use of message tags and a novel classifier construction scheme to implement its world model. Message tags are any part of a message that have conventionalized meanings within the classifier system (this is alluded to in section 3.2). For instance, in CFSC2, the input processor always produces messages with a prefix of "001" and the output processor only examines messages with a prefix of "111". The final bit in these tags corresponds to a distinction between current state and lookahead state. For example a message with a tag of "000" would be predicted input in case some particular classifier was activated. By use of these tags, classifiers are used as elements of the world model.

In CFSC2, two forms of strength are used. One is a static form, similar to the standard sort. The other dynamically accrues as the system executes. This

dynamic strength is based upon the predictive support given a classifier. This support disappears as the classifier is activated, and is not used in assigning genetic precedence[2].

CFSC2 has several advantages over the systems I demonstrate in this work. The classifier system modelled has internal messages and lookahead is integrated into the genetic algorithm. The systems I have demonstrated are, in a sense, like single state automata. They can react differently to different input, but must always (with the exception of any learning accomplished) react the same way with the same input. This is a major limitation which CFSC2 does not have.

The interface between the genetic algorithm and the classifier system is better constructed than that in the systems I have shown here. Certain genetic operations are triggered by events. When the automaton moves from one model state to the next, a classifier is constructed predicting that transition. This is how the world model is constructed.

Even though CFSC2 is clearly an impressive system, there are a few problems. The lookahead component produces transient information, so that much of the effort used in lookahead is lost. The scheme also requires an additional type of strength, which complicates the system. There is no lookahead for situations that are unrelated to the current situation, so if one is suddenly confronted with a situation, the lookahead will not have occurred. Finally, the scheme for combining lookahead information from different sources is primitive, with an implicit assumption that the information is independent.

These reasons lead one to wonder if there is not some way of combining the qualities of CFSC2 with that of Dyna-Q-CS. Replacing the bucket brigade used with a Q-learning based component should not be difficult. Getting the Dyna

---

[2]This is a problem with the bucket brigade. An economy where strength is conserved is not conducive to lookahead. Riolo solves this problem by making a new dynamic strength which is not conserved. Unfortunately, its dynamic nature makes it less useful than it might otherwise be. Q-learning does not have this problem.

architecture and integrating lookahead from different sources, however, might be difficult.

8.4 Improvements to Q-CS

The empirical results in all three worlds simulated show problems with the asymptotic performance of Q-CS. In N-World, the cause is clear. Once the world model internal to Q-CS is complete and accurate, the optimal performance can be gained by deterministically choosing the highest rated classifier. Q-CS does not do this. There are several options that might be tried.

The simplest option is merely to reduce the noise in the current policy of classifier selection. In the current implementation the policy is based upon the cube of the synthesized Q value. A quartic or higher order function would produce less noise in the selection metric.

Watkins [1989] uses a simulated annealing technique (also described in section 6.4). The probability of selecting a particular action i in state x is:

$$P(i|x) = \left[e^{\alpha Q_{xi}} \Big/ \sum_{j \in A_x} e^{\alpha Q_{xj}} \right].$$

The parameter, α, increases over time, so that differences in Q values become more and more important. As we have already described a method for synthesizing Q values, this equation could be adopted directly.

Sutton [1990] believes that the deterministic policy of choosing the highest valued action at each choice point should be adopted. His solution to the problem of exploratory behaviour is to place explicit value upon trying actions that have not been tried recently. Such a scheme could be adapted to Q-CS.

The techniques used in my analysis of results inspires another solution. Student's t-test provides a technique for determining the likelihood for the mean of a distribution to be significantly different from some value. It is this sort of significance that is important in determining which action to take. The choice of action should be nearly deterministic when there is a high significance to the difference in the mean utility between two potential actions. If, for each action available,

we create a t distribution, based upon the sample mean, standard deviation, and number of data points composing the utility of that action, we can choose a random value from each of those distributions. The action that produced the highest random value would be chosen. This takes into account the variance of utility values, and how much experience the system has with the action in question.

Another problem with Q-CS is the lookahead scheme. It inherits much of its architecture from La-Av. With a Dyna architecture, however, there is not as much need for other forms of lookahead. It would be interesting to try a purer form of Q-learning, where strength values are updated incrementally. This would be a better test of Q-learning within the classifier system domain. I think such a test should be made.

8.5 Improvements to La-Av

La-Av consists of many components. Several components are there which can be independently manipulated. One of the most obvious things to change is the method by which La-Av handles its competition for action selection. The highest rated individual that specifies an action is chosen as a sort of champion, to compete with the champions specifying other actions. Once the lookahead scheme evaluates the classifiers, there is no particular reason that action selection can not proceed as in the reconstructed Animat, with all classifiers specifying the same action adding their rating to the probabilities of that action being selected.

In the lookahead scheme itself, there are a number of things that might be changed. From the N-World data, it appears that La-Av puts too large a value upon the unknown. Something other than the arithmetic mean of all classifiers in the system might produce better results. For example, the geometric mean might be a better estimate. There is nothing particularly principled about using the geometric mean, but it would tend to reduce the effect of high value being placed on the unknown just because there are a few highly rated classifiers in the system.

Another area for potential improvement is in integration of reward and classifier rating data stored in the world model. At present, rewards are treated as if

they were at the same scale as classifier ratings. One could construct theoretical arguments that, given the 0.20 ratio for the bucket brigade, that reward values should be scaled up by a factor of five. The calculation is, however, complicated by the existence of taxes. Empirical tests using the factor of five scale up were not promising, but that does not mean that some form of compromise between the scales discussed would not improve system performance.

Finally, the lookahead scheme used, with its bags and values for the unknown, is also largely an orthogonal component. A scheme using followsets, or some other means of computing lookahead is totally compatible with the La-Av framework (although I suppose one would have to changer the name, since "Av" refers to the average scheme used to determine the value of the unknown).

8.6 Improvements to Dyna-Q-CS

Dyna-Q-CS has much potential for added reasoning capability. As currently implemented, it incorporates a two move lookahead. It also does dynamic planning by choosing classifiers at random, and updating the strengths of these classifiers based upon the world model.

One of the difficulties with Dyna-Q-CS that it inherits from Q-CS is that it runs very slowly. This is largely due to the two move lookahead incorporated in the evaluation function. In Q-CS, this was done in order to make good use of the world model. In Dyna-Q-CS, dynamic planning can take the place of lookahead. This would allow a return to the simpler update function used in Q-learning.

The choice of classifiers to be updated can incorporate many planning schemes. For example, by working backwards from classifiers that obtain rewards and updating those classifiers that have the rewarding classifiers in their followset, one can essentially plan by backward-chaining. Forward chaining, and other control strategies can be emulated as well. One of the advantages of placing such planning work under the Dyna paradigm is that partial plans need not be discarded, because the results of the planning are reflected in changes to the strengths of the

classifiers involved. The system does not have to wait to finish a plan if the world demands an immediate response.

8.7 Experimental Worlds

After performing this research, I am left with the (not too surprising) opinion that the performance of a situated autonomous learner is strongly dependent upon the actual world being investigated. In some sense, research in this area can be seen as determining which algorithms are better for what type of world, rather than just which algorithm is better. For instance, it is not at all clear that Octworld and Hexworld are particularly conducive to lookahead. These worlds are so ambiguous that it makes more sense to look back. That is, if there are all blank areas around now, and a particular action has been performed from another place that was surrounded by blank areas, this information can be used to disambiguate the current situation. Without this ability to look back (and none of the systems I have demonstrated here have this ability), looking ahead is not very easy or productive. I did not, however, choose Octworld and Hexworld to show off the abilities of my algorithms. I chose them as reasonable test cases for any system that claims to handle even simple worlds. I do not think these worlds were deceptive for my algorithms, but lookahead may not have been particularly useful.

In the real world, creatures are bathed in an apparently infinite variety of sensations. There are so many features to examine that it would not be hard to believe that an individual is never again faced with exactly the same sensations. This is in sharp contrast to Octworld and Hexworld, where many places offer exactly the same sensory input. In Octworld and Hexworld, the problem is determining which of many situations is the one currently occupied by the automaton. In the real world, the problem may be more of determining which, given that the sensory information always varies, situations are similar, rather than trying to determine which ones differ.

It is with that in mind that I would like to investigate more complex worlds. I am, however, left somewhat ambivalent about the use of complex worlds, either

simulated or real. To a certain extent, I think the most interesting data I have collected is from N-World. Nevertheless N-World is too simple to encompass a large number of the phenomena we would want a learning automaton to handle.

Another feature of the real world is its dynamic nature. All the worlds investigated were static. Action of the learning automaton did not affect the world (except in changing the placement of the automaton). Presumably, this is one of the reasons why the simulation ends when the automaton reaches a goal. Since the world would not change, the automaton would stay near the same goal and continuously receive rewards. A world with depletable resources and seasons might make strategies that succeed in a static world far from optimal.

Once worlds have depletable resources, competition may become an important factor. When several competing automaton must exist in the same world, complex strategies may be required. For example, the optimal solution to any finite Markov model is deterministic. Game theory suggests mixed strategies. A new world model may be required to handle competitive strategies.

8.8 Conclusion

While much research has been accomplished, there are still many unanswered questions. The poor performance of the lookahead classifier systems in Octworld and Hexworld suggest further changes to these algorithms. There are many parameters that can be varied, and the prognosis for improved performance is good. It would also be useful to form a broader range of test worlds in order to get a better idea of the relative strengths and weaknesses of the various algorithms for controlling situated learning automata.

Chapter 9

Conclusion

9.1 What Has Been Done?

This research into genetic algorithms and classifier systems has produced many novel ideas, as well as several computer programs to exploit them. The use of a followset to create an explicit and predictive world model for classifier systems is particularly important development, allowing more sophisticated control of classifier activation. The idea of using multiple classifiers to tease out extra information when deciding upon an action within a classifier system is also valuable. The extension of modified Q-learning to classifier systems offers an alternative to the bucket brigade. This alternative has been shown to support the extension of Dyna-Q-learning to classifier systems.

The research has also shown new results by reconstruction and modification of existing systems, such as Wilson's Animat, and comparing these results with the systems I have devised. In addition, it has also shown the difficulties of reconstruction where adequate information is not provided in the literature, such as in the case of Holland and Reitman's CS-1.

This research demonstrates some of the utility of an explicit world model. In a continuously learning system such as a classifier system, the world model is being constantly built and modified. Utilizing such an volatile model challenges current techniques. The incremental planning technique represented by Sutton's Dyna systems seems an appropriate framework. A lookahead system such as was

used in La-Av, or indeed the lookahead system used by Riolo [1990], is expensive to run, and the information gained is not used in later runs of the system. In a Dyna-like system, the strengths of the rules themselves are modified, so that future trials may make use of lookahead work done at a previous time. The creation of Dyna-Q-CS brings this capability into the classifier system domain.

9.2 Where Does This Lead?

The empirical results show that there are still considerable problems with Q-learning based classifier systems and lookahead classifier systems in general. There are many parameters that may still be changed, and types of world models that should be investigated. For improvements to the Q-learning based systems, further techniques from Q-learning are the obvious place to look in order to find improvements. It seems that some form of simulated annealing of the action selection scheme, rather than the fixed noise system currently incorporated might be effective in improving the asymptotic performance of the system. It appears that faster hardware might be required to make this investigation possible.

Since the main strength of systems with explicit world models is speed of learning, these systems should be tried in more dynamic environments. Exploring many differing worlds with these systems may help point out their strengths and weaknesses, as well as helping to classify the worlds investigated into some hierarchy. It is difficult to tell what qualities will be most important. Perhaps it is time to try such a system as an actual robot controller, so that the appropriate balance between exploring the world and exploiting this knowledge can be found.

9.3 Conclusion

I have made a point in this thesis, of making use of information other then reward information from the environment to guide situated autonomous learning. Part of the reason for this interest is the observation that the real environment does not produce reward information. If one programs a robot to journey to a particular

location, one can not rely on the environment to produce a reward. If the robot finds some point where its sensors indicate it has arrived at the goal, it might reward the strategy that got it to that position. The position that the robot reaches, however, may not be the actual location desired. The sensors of the robot (along with the software) may just be incapable of determining the difference between the location occupied, and the true goal. Rewards must be produced internally by the automaton. Rewards are not produced by the external environment.

The current design of classifier systems, while reasonably successful, fails to make use of a significant amount of the information available. This research demonstrates two ways to make use of some of the data currently ignored. An explicit world model allows learning ability even when no rewards are present. This information can be used to improve the performance of the system. The use of multiple classifiers to allow combination of information in a non-linear way also holds promise. Information can only be gained from the environment at some cost. Making better use of this information can limit these costs.

Appendix A

A Rational Reconstruction of Wilson's Animat and Holland's CS-1

A Rational Reconstruction of Wilson's Animat and Holland's CS-1

Gary Roberts
Department of Artificial Intelligence,
University of Edinburgh,
Scotland

Abstract

In the short history of Genetic Algorithms, there have been a plethora of techniques used. Even within the subfield of classifier systems, many differing implementations exist. It becomes difficult to compare ones results with others, and to determine the cause of actual performance differences. To that end I have attempted a rational reconstruction encompassing two systems described in the literature: Wilson's Animat and the CS-1 system of Holland and Reitman. The results obtained differ, sometimes markedly, from the published versions they attempt to duplicate.

1. Introduction

Research involves the investigation of new ideas. Part of the investigation is the comparison of a new technique with that of previous works. Most papers specify a problem domain, a proposed solution, and a measurement of the efficacy of the proposed solution. Unfortunately, such papers do not allow ease of comparison between various techniques, because the problem domains differ.

While there is something to be gained by establishing a fixed set of problems, this is not a complete solution. Some techniques may require particular problem spaces in which to excel, and there is always the problem of work that predates the establishment of these standards. These problems are exacerbated in a rapidly developing field such as Genetic Algorithms.

Rational reconstructions are generalisations of existing works [Bundy 1986]. Such rational reconstruction allows comparisons between works, within the common framework of the reconstructed problem and solution domain.

With this motivation, I have attempted a rational reconstruction including the works of Wilson [1985] and Holland [1978]. I chose the first work because it seemed a well documented fairly straight forward basic design. The latter was chosen on the basis of general availability. It appeared in a book of wide distribution, and had been published some time ago, so many people should find the work easily accessible.

While I intended to compare the two algorithms on each other's documented problems and compare results, my attempts to duplicate each algorithm's performance on its own problems varied sufficiently from the published results for these algorithms that such comparison would hardly be convincing. In the physical sciences, experiments must be reproducible to be considered discoveries, but it is also important for acceptance of validity that an experiment is independently verified. As is often the case in experimentation, my results both confirmed and conflicted with the published accounts. Since I have not imported the exact programs used in the published works, this paper in no way attempts to repudiate these works. In fact, the confirmation of comparative results without duplicating the actual results can be seen to strengthen the claims made in the articles. Still, the differences serve as warning that published articles rarely contain sufficient information for duplication, and trivial variations can possibly produce massive differences in results.

2. The Problem Domain

Genetic algorithms have been applied to learning automata in simulated environments. Holland

[1978] designed a complex system, and reported on a partial implementation. Similarly, Wilson [1985] has worked with a simple system. In order to facilitate comparison, I will describe them in a common framework.

The environment is discrete. It can be pictured as a network, with nodes representing valid positions, and edges representing adjacency. An automaton interacts with the environment in the following way: It can read the information in the node it occupies, and issue an (possibly vector valued) action code to the world simulator. In the actual programs implemented, the action value encoded a request to move along a particular edge (In Wilson's model, the request might not be honoured, in which case the automaton remains at its current position). Certain nodes are specified as goal nodes. When one of these is reached, the simulation is ended.

Both algorithms use the "Michigan" style classifier system (i.e. the genetic algorithm works on classifiers within, rather than between, classifier systems). The genetic algorithm is used to generate new classifiers, based on the performance ratings of the old classifiers. In both Wilson's and Holland's automata, genetic manipulation is performed only when the automaton reaches a goal node.

3. Wilson's Animat

Wilson's environment [Wilson 1985] is a simple two-dimensional world. A rectangular (18×58) grid is extended in the natural way to a topographicaly toroidal surface. Each node has eight edges, which naturally map to the eight semi-cardinal compass directions. Each node is describable by two bits (which Wilson describes as "food smell" and "opacity". As the automaton is presumed to be sensitive to the nodes surrounding it, each node is labeled with the bits (starting from North and preceding clockwise to form a sixteen bit string) descriptive of the eight nodes surrounding it.

As stated previously, the automaton operates in two cycles. In the first cycle, the classifiers guide the automaton to a goal. In the second, the genetic algorithm is applied to produce new classifiers, and delete old ones.

The classifiers consist of an eight position vector, each position drawn from the set {0, 1, #}, and an action specifier, taking the value of one of the eight possible directions of motion. Also associated with each classifier is a strength and distance value. The strength is a measurement of the utility of a classifier. The distance is an estimate of how far away the presumed goal.

In the first cycle, the automaton receives the information about the node. The set of classifiers with matching taxons is determined (If none match, a matching classifier is engineered on the spot, either at random, or using a rather ad hoc technique documented in Wilson's paper). Of these, each receives a rating (strength / distance), and one is selected (with probability rating / sum(ratings)). All classifiers (that match the input) that specify the same action (the current action set) are treated identically: they have a fraction (0.2) of their strength removed, and distributed equally to the previous action set.

In the second cycle, after a goal node is reached, the genetic algorithm may be invoked. Three genetic operations are supported: crossover, intersection, and duplication. A classifier is chosen for reproduction with probability proportional to its strength (distance is not considered). The actual mutation to use is chosen at random (0.5 chance of reproduction, 0.25 chance each of crossover or intersection). If the mutation is crossover or intersection, a second classifier is chosen from those with the same action (again with probability proportional to strength). Since the population of classifiers is to remain of constant size, a classifier is selected for deletion (with probability inversely proportional to strength).

Wilson's system solved the problem in an average of four or five steps after training. This was compared to an optimum of 2.2 steps average for an omniscient automaton. As Wilson's animat has a very limited sensory range, and can not explicitly store internal state information, this optimum seems unobtainably low. To put things in a more realistic perspective, I designed a set of classifiers and disabled any mechanism that would change this set. The result was a system that averaged success in 3.1 steps. While this may, or may not, be the optimum obtainable for a single state automaton with the sensors available, it at least forms an upper bound. It also serves as a reasonable goal for a learning

automaton with the same restrictions. Wilson's animat results are quite respectable.

Unfortunately, I was not able to reproduce the documented performance. After nine thousand iterations, the average performance of the reconstructed automaton over the next thousand iterations averaged 6.5 steps over twenty individual runs. The standard deviation was 0.6. Wilson's results are better by over two standard deviations.

There is another interesting potential difference between the results. Wilson reports no significant improvement in performance when the genetic algorithm is removed. While I have only one trial run at the moment, turning off the genetic algorithm in my own routines produced an average of 11.7 steps to solution after nine thousand iterations. The worst run using the genetic algorithm averaged 7.94 steps after training. Of course a singular event is not a scientific result. Still 11.7 is over eight standard deviations from the mean, and this avenue will be further investigated.

4. Holland and Reitman's CS-1

Holland [1978] describes a complex automaton, with multiple sets of classifier systems. A much simpler subset is actually implemented. The simplicity of the subset limits the automaton to rather simple worlds, at least as far as connectivity between nodes is concerned. There are several places where the work is vague, so it is not surprising that I have not managed to get the same results.

In Holland's design the classifier is somewhat more complex than Wilson's. The taxon is not only sensitive to the environment, but also to the last action of the automaton and internal messages. The action consisted of a single bit, for controlling an external action, and a message bit string, to put on a message list. The evaluation section consisted of four quantities: predicted payoff (in the implementation vector valued), age, frequency, and attenuation.

In the stimulus-response cycle, each classifier that matches the current situation is given a priority based on the number of non-wildcarded fields multiplied by "the amount of the current needs fulfilled by this classifier's predicted payoff" (The precise meaning of this quote is not clear to me). One of the top ten classifiers is then selected at random, based upon its relative priority. The selected classifier has its action performed, its age halved and its frequency incremented by one. If the predicted payoff of the selected classifier is less than that of the previously chosen classifier, the attenuation factor of each classifier chosen since the most recent goal achievement (excluding the currently selected classifier) is incremented.

The automaton learns only when a goal node is reached. First, a non-genetic learning scheme is applied. The classifiers that have been activated since the previous learning cycle have their predicted payoffs modified to reflect the achieved goal. "Those predicted payoffs that were consistent with (not greater than) this reward are maintained or increased; those that overpredicted are significantly reduced, all according to their attenuations." All attenuation factors are reset to zero.

The creation of new classifiers is accomplished by the crossover mutation. The parents are selected each with "probability proportional to its predicted payoff". The new classifier has predicted payoff and effector settings from one parent, and an age that is the average of the two parents. Attenuation factors and frequency are set to zero. In order to make room for the new classifier, an old classifier is deleted. From the classifiers with the greatest age, the one with a taxon most closely matching the new classifier is chosen for deletion.

In Holland's work, the use of genetic learning significantly increased the learning speed (and possibly the asymptotic results as well) of the learning automaton, both in an initial learning case, and in the case of an experienced automaton in a new environment.

In reconstructing Holland's work, several questions must be answered. There are several important ones outstanding, as well as a concern for what I believe to be a problematic case. One must determine the "current needs fulfilled" given a payoff vector. A comparison function for payoff vectors is needed. A mapping from payoff vectors to scalars must be made. A function must be designed to modify a classifier payoff vectors, "all according to their attenuations". It also seems that according to the algorithm described, a classifier might not be strongly affected by a particular payoff, even if that

classifier was the last to be used to obtain the payoff. The reason is that it might have been used earlier, and thus have a very high attenuation value.

In order to create a "current needs fulfilled" value, the automaton is considered to have a needs vector (similar to, but opposite in magnitude from the original article's resource vector). Each element of the vector was incremented after an action was performed by the automaton. The "current needs fulfilled" is derived by summing the minima of corresponding positions in the needs and payoff vectors. When a goal node is achieved the reward for obtaining a goal is subtracted from the corresponding need. If the result is less than zero, the need is set to zero.

The comparison function on payoff vectors, for purposes of determining attenuation factors, is a simple comparison between each element in the payoff vectors. If any value in the currently selected classifiers payoff vector is less than the corresponding element in the previously selected classifier's payoff vector, the attenuation factors of all previously selected classifiers (since the last achieval of a goal state) have their attenuation factors increased. Note that this comparison function is not a partial ordering.

A mapping of payoff vectors to scalars is needed in order to select parents, for new classifier construction, with "probability proportional to their predicted payoff". The function chosen was simple summation of the payoff vector.

Modification of payoff vectors was accomplished in the following manner. For each element in the payoff vector, a weighted average is computed (((frequency-1) $\times$ payoff + reward) / frequency). If this average is greater than the payoff, then the payoff is increased (to ((attenuation + 1) $\times$ payoff + average) / (attenuation + 1)), otherwise it is decreased (to the weighted average).

As stated earlier, I was a bit concerned with the problem of classifiers that had been selected earlier having large attenuation factors, even though they had reselected in the recent pass. Because of this concern, the results given here are based on a scheme where attenuation factors are kept for each selection of a classifier. Payoff vectors were modified as above, with the most recently selected classifier updated first.

The criteria for success used by Holland is a bit more involved than that of Wilson, due to the needs of the automaton being vector defined. Since achieving a goal might only satisfy one of several needs, it is not enough to give the average number of steps to a goal. Holland solves this by giving a specific set of conditions for success, based upon the actual nature of the world simulated. There are two goal nodes, each providing a different one of two needs. One goal provides at twice the rate of the other. Holland's criteria is ten consecutive runs to a goal, where the minimum possible steps to a goal is taken, and the one goal (presumably the one with the least resource) is visited twice as often as the other. The better the learning algorithm, the fewer iterations before this occurs.

For Holland's environment A, Holland's automaton achieved the success criteria after 212 iterations. Attempted without the genetic algorithm, it took 2161 iterations.

My results, in the same environment, led to the following observation. As the reward at each goal node was amply sufficient to satiate the needs of the automaton, there was no preference to move to one goal rather than another. Random variation eventually produced a six to four run (roughly two to one out of ten) after 6363 iterations, while the first ten consecutive optimum runs to goal (in a one to one ratio) occurred after 839 iterations. This optimum performance was repeated 19 times (once in a four to six ratio, all other in a one to one ratio) before Holland's criteria were met. Without the genetic algorithm my reconstructed automaton did not even get ten consecutive optimal runs after ten thousand iterations.

5. Conclusions

The difficulty of comparing various published techniques remains formidable. Even the best documented algorithms can defy duplication. It remains for researchers to do the best they can in providing all of the salient information. Making the code available as well is also useful, although differences in programming languages tend to limit ease of transport.

The results obtained generally confirm the utility of the genetic algorithm in solving problems within the specified domain. The differences in detail are quite marked. I hope to be able to more closely replicate these works before I compare them with each other, or with the results of my own research into improved classifier systems.

Acknowlegements

I would like to thank my supervisors for their help and guidance: Henry Thompson and Roberto Desimone. I would also like to thank Stewart Wilson for his aid in identifying some of the differences between earlier versions of my program and his reported work. The extant differences between my work and others, as well as flaws in this work, are my responsibility alone. The equipment used for this project, as well as for document preparation, was donated by the Rank Xerox Ltd. University Grants Programme.

Bibliography

Bundy, Alan (1986) *What Kind of Field is Artificial Intelligence?*, DAI Research Paper № 305, Department of Artificial Intelligence, University of Edinburgh, U.K.

Holland, John and J. Reitman (1978) "Cognitive Systems Based on Adaptive Algorithms", *Pattern-Directed Inference Systems*, Waterman, D. and Frederick Hayes-Roth (Eds.), Academic Press, Inc., New York, U.S.A.

Wilson, Stewart (1985) "Knowledge Growth in an Artificial Animal", *Proceedings of an International Conference on Genetic Algorithms and their Applications*, John Grefenstette (Ed.), held June 1985 at Carnegie-Mellon University, Pittsburgh, U.S.A.

Bibliography

Albers, G., Brand, H., and Cellerier, G. (1985). A microworld for genetic artificial intelligence. memo NR. 1, Genetic Artificial Intelligence and Epistemics Laboratory, University of Geneva, CH 1211 Geneva 4 Switzerland.

Antonisse, J. (1989). A new interpretation of schema notation that overturns the binary encoding constraint. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86–91, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Belew, R. K. and Gherrity, M. (1989). Back propagation for the classifier system. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 275–281, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Booker, L. B. (1982). *Intelligent Behavior as an Adaptation to the Task Environment.* PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, U.S.A.

Booker, L. B. (1985). Improving the performance of genetic algorithms in classifier systems. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 80–92, Held June 1985 at Carnegie-Mellon University, Pittsburgh, PA, U.S.A.

Booker, L. B. (1988). Classifier systems that learn internal world models. *Machine Learning*, 3:161–192.

Booker, L. B. (1989). Triggered rule discovery in classifier systems. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic*

Algorithms, pages 265–274, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Bridges, C. L. and Goldberg, D. E. (1989). A note on the nonuniform Walsh-Schema transform. Technical Report TCGA Report No. 89004, The Clearinghouse for Genetic Algorithms, University of Alabama. Tuscaloosa, AL 35487-2908, U.S.A.

Brooks, R. A. (1986). Achieving artificial intelligence through building robots. Technical Report A. I. Memo 899, Massachusetts Institute of Technology, Boston, MA, U.S.A.

Bundy, A. (1986). What kind of field is artificial intelligence? Research Paper 305, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, U.K.

Carbonell, J. G. and Hood, G. (1986). The world modelers project: Objectives and simulator architecture. In Mitchell, T. M., Carbonell, J. G., and Michalski, R. S., editors, *Machine Learning: A Guide to Current Research*, pages 29–34. Kluwer Academic Publishers, Boston, MA, U.S.A.

Carbonell, J. G., Michalski, R. S., and Mitchell, T. M. (1983). An overview of machine learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 3–23. Tioga Publishing Company, Palo Alto, CA, U.S.A.

Darwin, C. R. (1859). *The Origin of Species by Means of Natural Selection.* Penguin Books, London, UK.

Davis, L. (1989). Mapping neural networks into classifier systems. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 375–378, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Dawkins, R. (1989). *The Extended Phenotype.* Oxford University Press, Oxford, UK.

Dewdney, A. K. (1985). Computer recreation. *Scientific American*, 253(4):17–21.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence Through Simulated Evolution.* John Wiley and Sons, Inc., New York, NY, U.S.A.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10:447–474.

Goldberg, D. E. (1985). Dynamic system control using rule learning and genetic algorithms. In Joshi, A., editor, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 588–592, Los Altos, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Goldberg, D. E. (1989a). *Genetic Algorithims in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., Reading, MA, U.S.A.

Goldberg, D. E. (1989b). Genetic algorithms and Walsh Functions: Part I, a gentle introduction. *Complex Systems*, 3:129–152.

Goldberg, D. E. (1989c). Genetic algorithms and Walsh Functions: Part II, deception and its analysis. *Complex Systems*, 3:153–171.

Grefenstette, J. J. (1988). Credit assignment in genetic learning systems. In *Proceedings AAAI-88 Seventh National Conference on Artificial Intelligence*, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, U.S.A.

Holland, J. H. (1983). Escaping brittleness. In *Proceedings International Workshop on Machine Learning*, pages 92–95, Monticello, IL, U.S.A.

Holland, J. H. and Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In Waterman, D. and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems*, pages 313–329. Academic Press, Inc., New York, NY, U.S.A.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Inc, Reading, MA, U.S.A.

Huang, D. (1989). The context-array bucket-brigade algorithm: An enhanced approach to credit-apportionment in classifier systems. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 311–316, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Kaelbling, L. P. (1989). A formal framework for learning in embedded systems. In Spatz, B., editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 350–353, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Koza, J. R. (1990). Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer, J.-A. and Wilson, S. W., editors, *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 366–375, Cambridge, MA, U.S.A. The MIT Press.

MacDonald, A. (1986). Prediction and control in an active environment. In Mitchell, T. M., Carbonell, J. G., and Michalski, R. S., editors, *Machine Learning: A Guide to Current Research*, pages 183–187. Kluwer Academic Publishers, Boston, MA, U.S.A.

Riolo, R. L. (1990). Lookahead planning and latent learning in a classifier system. In Meyer, J.-A. and Wilson, S. W., editors, *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 316–326, Cambridge, MA, U.S.A. The MIT Press.

Roberts, G. A. (1989). A rational reconstruction of Wilson's Animat and Holland's CS-1. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 317–321, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Rosenschein, S. J. (1985). Forman theories of knowledge in AI and robotics. Technical Note 362, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, Menlo Park, CA 94025, U.S.A.

Rosenschein, S. J. and Kaelbling, L. P. (1987). A synthesis of digital machines with provable epistemic properties. Technical Report CSLI-87-83, SRI In-

ternational, Artificial Intelligence Center, Computer Science and Technology Division, Menlo Park, CA 94025, U.S.A.

Sammut, C. and Hume, D. (1986). Learning concepts in a complex robot world. In Mitchell, T. M., Carbonell, J. G., and Michalski, R. S., editors, *Machine Learning: A Guide to Current Research*, pages 291–294. Kluwer Academic Publishers, Boston, MA, U.S.A.

Smith, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. In Bundy, A., editor, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 422–425, Los Altos, CA, U.S.A. William Kaufmann, Inc.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning.*

Walpole, R. E. (1982). *Introduction to Statistics.* Macmillan Publishing Co, Inc., New York, NY, U.S.A., third edition.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, Cambridge, Cambridge, U.K.

Westerdale, T. H. (1985). The bucket brigade is not genetic. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 45–59, Held June 1985 at Carnegie-Mellon University, Pittsburgh, PA, U.S.A.

Westerdale, T. H. (1989). A defense of the bucket brigade. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 282–290, San Mateo, CA, U.S.A. Morgan Kaufmann Publishers, Inc.

Westerdale, T. H. (1990). Quasimorphisms or queasymorphisms? modeling finite automaton environments. In *Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, Bloomington, IN, U.S.A.

Wilson, S. W. (1985). Knowledge growth in an artificial animal. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic*

Algorithms and their Applications, pages 16–23, Held June 1985 at Carnegie-Mellon University, Pittsburgh, PA, U.S.A.

Wilson, S. W. (1990). The animat path to AI. In Meyer, J.-A. and Wilson, S. W., editors, *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 15–21, Cambridge, MA, U.S.A. The MIT Press.

Wright, C. E. (Unpublished). A computer model of a cognitive adaptive system. University of Michigan, Ann Arbor, MI 48109, U.S.A.